B

ANALIZA SPRAWNOŚCI ALGORYTMÓW

Znane są tysiące sposobów zabijania czasu,
ale nikt nie wie, jak go wskrzesić.
Albert Einstein

Komputery w powszechnym mniemaniu uchodzą za uosobienie szybkości. Nierzadko przecież zdarza się słyszeć, że oto pokonana została kolejna bariera prędkości obliczeń, a nowo zbudowany superkomputer w sekundę poradzi sobie z zadaniem, z którym cała ludzkość musiałaby się biedzić przez setki milionów lat. W takich sytuacjach laicy czasem zadają sobie pytanie, czy istnieją jeszcze dla komputerów jakieś niewykonalne zadania, których nie rozwiązałyby w mgnieniu oka. Wydawałoby się, że takich wyzwań już nie ma.

Nasz entuzjazm dla osiągnięć techniki musi jednak przygasnąć, jeżeli przydarzy nam się typowa przecież sytuacja, gdy musimy oczekiwać na uruchomienie programu na swym osobistym komputerze. Albo na wyszukanie określonych plików na jego dysku twardym. Albo na ściągnięcie kilkumegabajtowego pliku przez zapchane łącze internetowe. Albo na wyrenderowanie gotowej sceny w aplikacji do modelowania 3D. Albo…, albo… - przykłady można mnożyć w nieskończoność. Zatem nasze komputery nie są wcale takie szybkie. Czy można coś na to poradzić?…

Intuicja podpowiada nam, że tak. Faktycznie - możemy przecież zakupić szybszy dysk twardy, zafundować sobie lepsze połączenie z Internetem, wymienić procesor na nowszy, postarać się o lepszą kartę graficzną, i tak dalej. Wszystko to możemy zrobić my - **użytkownicy**, podążając trasą niekończącego się wyścigu technologicznego.
A co mogą zrobić twórcy aplikacji, czyli **programiści**? Przecież on nich również zależy szybkość działania ich produktów: nawet najszybszy dysk twardy będzie bowiem pracował nieefektywnie, jeżeli zainstaluje się na nim mało wydajny system plików; najlepsza karta graficzna może sobie nie poradzić z rysowaniem świata gry trójwymiarowej, jeśli będzie zmuszona do jego całościowego przetwarzania dla każdej wygenerowanej klatki; wreszcie, nawet najnowszy procesor może się ugiąć się pod ciężarem skomplikowanych operacji na ogromnym zbiorze danych. Dlatego programiści muszą dbać o odpowiednią **optymalizację** działania swoich wytworów, a szczególnie tych ich części, które są najintensywniej wykorzystywane przez użytkowników.

Optymalizacja jest aczkolwiek trudnym zadaniem, które można wykonywać na wielu płaszczyznach. Możliwe jest optymalizowanie projektu aplikacji, określającego jej nadrzędną strukturę - jak choćby klasy i ich składowe. Źle zaprojektowany program ma bowiem wszelkie szanse, by działać jeśli nie całkiem niepoprawanie, to „przynajmniej” bardzo nieefektywnie.
Drugą stroną optymalizacji jest dobieranie odpowiednio **szybkich algorytmów** do realizacji chociaż tych najbardziej newralgicznych zadań. Efektywny algorytm może bowiem skrócić czas ich wykonywania setki, tysiące, a nawet miliony (!) razy, produkując jednocześnie identyczne wyniki.

OK, być może w tej chwili nieco przesadzam. Nie ulega jednak wątpliwości, że dla każdego niemal zadania istnieją algorytmy lepsze i gorsze, działające szybciej i wolniej, krótsze i dłuższe w zapisie oraz łatwiejsze i trudniejsze w implementacji - a wszystkie one są tak samo **poprawne** w sensie generowanych rezultatów.
Dla wygody programistów najważniejsze byłoby zapewne kryterium prostoty, jednak nie zapominajmy, że aplikacje tworzymy raczej dla użytkowników innych niż my sami. Smutną prawdą jest fakt, że takich „postronnych" osób niewiele interesują upodobania autora programu czy nawet jakość produktu z punktu widzenia inżynierii oprogramowania; dla nich ważniejsze jest bowiem to, co mogą zobaczyć i odczuć bezpośrednio: wygodny interfejs użytkownika, rozbudowane możliwości czy nareszcie szybkość działania.

W interesie popularności naszych dzieł leży więc (między innymi) wybór odpowiednio efektywnych algorytmów, poprzez które aplikacje będą realizowały swoje cele. Jak jednak ocenić, który algorytm jest szybszy? Czy istnieją ścisłe kryteria wyznaczania sprawności danego algorytmu?…
Są to bardzo słuszne i ważne pytania, na które postaram się tutaj odpowiedzieć.

Złożoność obliczeniowa

Kiedy mamy na myśli efektywność wykonania jakiegoś zadania, łatwo możemy posługiwać się miarą czasową. Zrobienie czegoś w 10 minut jest bardziej efektywne niż zrobienie tego samego w kwadrans, nie mówiąc już o półgodzinnej czy godzinnej pracy. Czy jednak podobne kryterium może się stosować do algorytmów?

Wyobraźmy sobie, że mamy do wyboru dwa algorytmy realizujące ten sam cel i produkujące identyczne wyniki, ale napisane przez dwie różne osoby. Jedna z tych osób twierdzi, że jej algorytm jest szybki, bo wykonał się w 41 sekund; druga utrzymuje, że jej algorytmowi zajęło to tylko 29 sekund. Czy znaczy to bynajmniej, że ten drugi sposób jest szybszy?…
Otóż niezupełnie, bowiem niemal na pewno obie osoby uruchamiały swoje algorytmy w **różnych warunkach**. Aby więc obiektywnie porównywać ich sprawność, należałoby te warunki znać - tzn. wiedzieć:

- Jakie komputery zostały użyte w przeprowadzonych próbach?
- Jakie procesory posiadały?
- Pod kontrolą jakich systemów operacyjnych pracowały?
- Czy napisane programy działały w trybie wyłączności, a jeśli nie, to jaki miały priorytet?
- W jakich językach zostały napisane oba programy?
- Jakich kompilatorów użyto do ich skompilowania?
- Czy w owych kompilatorach były włączone opcje optymalizacji?
- itp. itd.

Jak widać, potrzebnych informacji jest całe mnóstwo, zaś nawet posiadanie ich wszystkich nie upewnia nas, że czegoś nie przeoczyliśmy. Poza tym mając tak szeroki zasób wiadomości, porównywanie sprawności obu algorytmów wcale nie staje się prostsze, a w praktyce jest prawie **niemożliwe**.

Do całego problemu trzeba zatem podejść zupełnie inaczej. Przede wszystkim należy uświadomić sobie, że algorytm to nie jest skompilowany i funkcjonujący program (lub jego część), lecz pewien przepis, ogólny ciąg kroków. Co najważniejsze, jest on **niezależny** od wszystkich warunków „technicznych", wymienionych powyżej - nawet od kompilatora i języka programowania. Ten sam algorytm może być przecież zapisany w każdym niemal języku; popatrzmy chociażby na kod poszukujący danego elementu tablicy jednowymiarowej, zapisany w czterech językach programowania:

```
// C(++)
int Szukaj(const int* pTablica, unsigned uRozmiar, int nSzukany)
{
   for (int i = 0; i < uRozmiar; ++i)
         if (pTablica[i] == nSzukany)
               return i;

   return -1;
}

// Object Pascal (Delphi)
function Szukaj(const ATablica : array of Integer; ASzukany : Integer)
               : Integer;
var
   i : Integer;
begin
   for i := 0 to Length(ATablica) - 1 do
   begin
         if ATablica[i] = ASzukany then
         begin
               Result := i;
               Exit;
         end;
   end;

   Result := -1;
end;

' Visual Basic
Function Szukaj(Tablica() As Integer, Szukany As Integer) As Integer
   Dim i As Integer

   For i = 0 To Len(Tablica) - 1
         If Tablica(i) = Szukany Then
               Szukaj = i
               Exit Function
         End If
   Next i

   Szukaj = -1
End Function

// PHP
function Szukaj($aTablica, $nSzukany)
{
   foreach ($aTablica as $idxIndeks => $nWartosc)
         if ($nWartosc === $nSzukany)
               return $idxIndeks;

   return -1;
}
```

Możliwe jest nawet więcej: algorytm można przecież zapisać, nie używając do tego **żadnego języka programowania**, lecz posługując się tylko pseudokodem:

```
Funkcja Szukaj(Tablica[] :int, Szukany :int) :int
   i :int

   Dla Każdego i := Indeks(Tablica) Wykonaj
         Jeżeli Tablica[i] = Szukany To
```

```
                Zwróć i
            Koniec
        Koniec

        Zwróć -1
    Koniec
```

W takim wypadku wszelkie rzeczywiste miary, dotyczące faktycznego czasu wykonywania algorytmu tracą jakikolwiek sens. Potrzebujemy zatem takiego oszacowania, które pozwoli wyznaczyć efektywność algorytmu nie tylko bez jego kompilacji i uruchamiania, ale nawet bez zapisywania go w żadnym istniejącym języku programowania. Miara efektywności powinna bowiem dotyczyć tylko abstrakcyjnego **ciągu kroków**, jakim jest każdy algorytm.

Dla zapewnienia ścisłości i wygody czytelników, a także swojej własnej, wszystkie użyte dalej algorytmy będę jednak zapisywał w języku C++.

Klasa algorytmu

Zdecydowaliśmy więc, że nie będziemy się zajmować rzeczywistym czasem działania algorytmu na jakimś komputerze, lecz **ilością elementarnych kroków**, jakie musi on wykonać, aby wywiązać się ze zleconego mu zadania. Za elementarny krok uważamy natomiast pojedynczą, prostą instrukcję; przyjęło się zresztą, iż w analizie sprawności algorytmów bierze się pod uwagę **głównie instrukcje porównania**, ewentualnie przypisania.

Teraz trzeba sobie zadać pytanie: czy ilość owych elementarnych kroków będzie w każdym przypadku taka sama? Nietrudno domyślić się, że nie. Algorytmy tworzymy przecież po to, aby operowały one na nieznanych z góry danych, zatem pracochłonność wykonania czynności algorytmu może ściśle zależeć od tych danych. Dokładniej - może ona zależeć od **rozmiaru** wejściowych parametrów algorytmu.
Pojęcie rozmiaru jest tu użyte bardzo ogólnie, a jego dokładne znaczenie jest nierozerwalnie związane z konkretnym zagadnieniem, czyli rozważanym algorytmem. Dla przykładowej procedury przeszukiwania tablicy rozmiarem danych będzie oczywiście ilość elementów tej tablicy; przy sprawdzaniu, czy podana liczba jest pierwsza, decydującą rolę odegra ona sama; podczas znajdowania pozycji jednego napisu wewnątrz innego rozmiar danych jest wypadkową długości zarówno przedmiotu, jak i zakresu poszukiwań; i tak dalej. Można więc stwierdzić, że:

W analizie efektywności algorytmów **rozmiar danych** jest tą wielkością opisującą wejściowe dane dla algorytmu, która najbardziej **wpływa na ilość kroków** podjętych przy rozwiązywaniu problemu.

Czas wykonywania algorytmu, liczony liczbą elementarnych kroków, najczęściej nie będzie więc wielkością stałą, lecz **funkcją rozmiaru danych** wejściowych - funkcją w rozumieniu matematycznym. Szacowanie efektywności algorytmu polega zatem na znalezieniu owej funkcji i tym się właśnie teraz zajmiemy.

Znajdujemy złożoność praktyczną

Jako przykład weźmiemy sobie stosunkowo prosty algorytm sortowania, znany jako **sortowanie przez wstawianie** (ang. *insertion sort*). Być może znasz sposób jego działania - a jeśli tak, to zapewne wiesz również, że charaktertyzuje się on nieszczególną efektywnością. Skądkolwiek czerpiesz tą wiedzę, możesz ją teraz zweryfikować.

Przykład: sortowanie przez wstawianie

Najpierw powiedzmy sobie coś o samym algorytmie. Sortowanie przez wstawianie jest prostym sposobem na uporządkowanie tablicy dowolnych elementów. Oczywistym warunkiem jest istnienie jakiegoś kryterium możliwego uporządkowania (tzw. porządku liniowego) wśród elementów tablicy. W praktycznej sytuacji mogą to być złożone zasady - szczególnie jeśli sortujemy np. rekordy w bazie danych - ale dla nas nie ma to żadnego znaczenia. Liczy się sama możliwość ustalenia, który element jest mniejszy, a który większy; dlatego też celem pominięcia takich „technicznych" szczegółów będziemy zajmowali się wyłącznie sortowaniem liczb całkowitych typu `int`. Przy użyciu własnych typów danych i przeciążania operatorów można zresztą w niemal automatyczny sposób uzyskać algorytm dla dowolnego rodzaju elementów.

Spójrzmy więc na ową procedurę, sortującą tablicę o podanym rozmiarze:

```
void InsertionSort(const int* aTablica, unsigned uRozmiar)
{
   unsigned i, j;
   int nElement;

   // pętla zewnętrzna, wybierająca po kolei każdy element
   // (począwszy od drugiego, czyli tego o indeksie 1)
   for (i = 1; i < uRozmiar; ++i)
   {
         nElement = aTablica[i];

         // pętla wewnętrzna ma za zadanie stworzyć miejsce dla
         // naszego elementu
         for (j = i - 1;
               i >= 0 && aTablica[j] > nElement; --j)
               // czyni to, przesuwając elementy do przodu
               aTablica[j + 1] = aTablica[j];

         // a gdy ono już jest, trzeba zapisać element na tym miejscu
         aTablica[j + 1] = nElement;
   }
}
```

Zasada jej działania jest prosta. Zewnętrzna pętla `for` przebiega po wszystkich elementach tablicy, natomiast wewnętrzna zajmuje się szukaniem (albo raczej tworzeniem) właściwego miejsca dla aktualnego elementu. Robi to, przesuwając w stronę końca tablicy wszystkie liczby, które są większe od tej rozważanej (czyli `nElement`). Na powstały w ten sposób wakat wstawiana jest rzeczona liczba, a wówczas zewnętrzna pętla zajmuje się kolejnym elementem. Ten jest znowu porównywany z poprzedzającymi go liczbami, wstawiany na odpowiednie miejsce… i tak dalej, aż do końca tablicy.

Ustalamy reguły

Spróbujmy teraz zająć się sednem sprawy, czyli przybliżeniem czasu działania algorytmu. Jak wspominałem na początku, interesować nas będzie czas wyrażony w postaci **liczby elementarnych kroków** wykonanych przez procedurę.

Co można rozumieć przez to pojęcie? Na zwyczajnym komputerze, dokonującym naraz co najwyżej jednej czynności, za elementarny krok wygodnie jest przyjmować pojedynczą instrukcję - zwykle wiersz kodu. Trzeba jednak uważać (zwłaszcza w językach wysokiego poziomu), by nie robić tego bezmyślnie. Wywołania funkcji nie można bowiem traktować jako jeden krok, bo jej wykonanie zajmuje w rzeczywistości więcej pracy.
Najrozsądniej jest zatem uważać za pojedyncze kroki najprostsze instrukcje. Należą do nich głównie **przypisania** oraz **porównania**. Dla uproszczenia można jednocześnie

założyć, że obie te operacje zajmują tyle samo czasu - wówczas nie będzie potrzeby ich rozróżniania.

Drugą ważną kwestią jest sprecyzowanie, czym jest dla nas rozmiar danych wejściowych. Myślę, że w tym przypadku trudno o jakiekolwiek wątpliwości. Skoro przedmiotem naszych zainteresowań jest tablica, logicznym określeniem rozmiaru danych jest wielkość tej tablicy. Ściślej mówiąc, będzie to liczba jej elementów, a więc wartość zmiennej `uRozmiar`. Dalej będziemy ją oznaczać w skrócie jako *n*.

Celem naszych poszukiwań jest wobec tego odszukanie funkcji *f*(*n*), której wartości byłyby ilościami kroków algorytmu potrzebnych do posortowania *n*-elementowej tablicy. Możemy teraz głęboko odetchnąć i zabrać się do pracy…

Przyglądamy się algorytmowi

Zanim ustalimy ilość kroków algorytmu w zależności od rozmiaru danych, musimy wyróżnić te instrukcje, których wykonanie będziemy uważać za jeden krok. Zgodnie z ustaleniem z poprzedniego paragrafu, będą to:

- instrukcja inicjująca licznik (`i = 1`) na początku zewnętrznej pętli `for`
- sprawdzenie wartości tegoż licznika (`i < uRozmiar`) na początku każdego cyklu zewnętrznej pętli `for`
- przypisanie `nElement = aTablica[i]` w zewnętrznej pętli
- instrukcja inicjująca licznik (`j = i - 1`) na początku wewnętrznej pętli `for`
- instrukcja sprawdzające wartość tego licznika (`i >= 0 && aTablica[j] > nElement`) na początku każdego cyklu wewnętrznej pętli `for` (właściwie mamy tutaj dwa porównania, ale dla uproszczenia potraktujemy to jako jedną instrukcję)
- przypisanie `aTablica[j + 1] = aTablica[j]` w wewnętrznej pętli
- dekrementacja licznika (`--j`) pod koniec cyklu wewnętrznej pętli
- przypisanie `aTablica[j + 1] = nElement` w zewnętrznej pętli
- inkrementacja licznika (`++i`) pod koniec cyklu zewnętrznej pętli

Zakładamy, że pojedyncze wykonanie każdej z tych instrukcji trwa taki sam okres czasu, a koszt wykonania możemy oznaczyć po prostu jako 1. Pozostaje jeszcze kwestia ustalenia, jak często (w zależności od *n*) wykonuje się każda instrukcja.

Nie byłoby to bardzo trudne, gdyby pewna kwestia, której, jak się zdaje, nie sposób obejść. Chodzi o mianowicie o wewnętrzną pętlę: nie możemy bowiem dokładnie ustalić liczby jej cykli, bowiem nie zależy ona tylko od wielkości tablicy. Przeciwnie, przy szukaniu miejsca dla `i`-tego elementu liczy się **każdy z poprzedzających** go `i` - 1 elementów tejże tablicy. Aby ustalić ustalić dokładną ilość wykonanych przypisań `aTablica[j + 1] = nElement` trzebaby zatem… samodzielnie ją policzyć! To zdecydowanie niepraktyczne - cóż więc z tym zrobić?
Otóż na razie odłożymy sobie ten problem na półkę. Wprowadzimy po prostu dodatkowy symbol t_i na oznaczenie **liczby sprawdzeń** warunku wewnętrznej pętli `for` w zależności od wartości `i`. Później zastanowimy się, jak możnaby tę dodatkową zmienną usunąć.

Funkcja złożoności

Teraz należy ponownie przyjrzeć się procedurze `InsertionSort()` i oznaczyć łączny koszt z instrukcji spośród tych z listy podanej powyżej. Pamiętając o parametrach *n* i t_i, możemy to uczynić w ten sposób:

```
void InsertionSort(const int* aTablica, unsigned uRozmiar)
{
   // (wszystkie nieważne szczegóły pominięto)

   for (i = 1;                                    // 1
```

```
        i < uRozmiar;                                   // n
        ++i)                                            // n - 1
    {
      nElement = aTablica[i];                           // n - 1

      for (j = i - 1;                                   // n -1
           i >= 0 && aTablica[j] > nElement;            // sum_{i=2}^{n} t_i

           --j)                                         // sum_{i=2}^{n} (t_i - 1)

        aTablica[j + 1] = aTablica[j];                  // sum_{i=2}^{n} (t_i - 1)

      aTablica[j + 1] = nElement;                       // n - 1
    }
}
```

Koniecznie przeanalizuj ten przykład, aby zrozumieć, dlaczego koszty tych instrukcji są właśnie takie. Przy liczeniu cyklów pętli warto pamiętać, że wartość zmiennej `uRozmiar` to nic innego, jak nasze *n*.

W tym momencie z łatwością możemy już przedstawić sumaryczny koszt całego algorytmu. Dodając do siebie koszty wykonania poszczególnych instrukcji otrzymamy ostatecznie[1]:

$$T(n) = 4n - 2 + 2\sum_{i=2}^{n} t_i$$

Ta funkcja $T(n)$ nosi nazwę **złożoności praktycznej** algorytmu.

Złożoność praktyczna jest funkcją, która dla podanego **rozmiaru danych** wyznacza **dokładną** liczbę elementarnych kroków potrzebnych do wykonania danego algorytmu.

Popadamy w pesymizm

Nasza funkcja wygląda stosunkowo zgrabnie, ale ma jeden mankament: jest zależna nie tylko od *n*, ale też od t_i, czyli od faktycznego rozmieszczenia danych (elementów sortowanej tablicy). Wynika stąd, że nawet dla tablic tego samego rozmiaru czasy wykonania procedury `InsertionSort()` mogą się różnić. Obiecałem ci, że pozbędziemy się tego zgrzytu, więc pora to zrobić.

W praktycznych zastosowaniach każdy algorytm jest wywoływany wielokrotnie, najczęściej dla różnych danych. Dla każdego zestawu istnieje oczywiście jego własna wartość $T(n)$, zależna od *n* oraz od współczynników t_2, t_3, …, t_n. Pomyślmy jednak, czy ma ona jakiś praktyczny sens?… Potencjalnych zestawów danych jest nieskończenie wiele, więc interesowanie się złożonością algorytmu dla każdego z nich raczej mija się z celem.
Zamiast tego lepiej jest postawić na jakieś uogólnione przypadki, dla których wartości $T(n)$ będą charakterystyczne. Dlatego też w algorytmice rozważa się trzy takie warianty:

- przypadek **optymistyczny**, który oznacza najmniejszą liczbę wykonanych kroków
- przypadek **średni**, oznaczający złożoność algorytmu dla typowego zestawu danych
- przypadek **pesymistyczny**, odnoszący się do zestawu danych powodującego najdłuższy czas wykonania

[1] Tzn. po dokonaniu kilku przekształceń upraszczających sumy.

W każdym z nim zakładamy, że *n* jest takie samo (bo nadal chcemy, by od niego zależała złożoność), jednak wybranie któregoś przypadku determinuje wartości t_i. Innymi słowy, czynimy wówczas pewne (rozsądne) założenia o rozmieszczeniu danych wejściowych. W naszym przypadku chodzi o układ liczb w sortowanej tablicy.

Zacznijmy zatem od przypadku optymistycznego. Dla sortowania będzie nim fakt, iż podana tablica jest już rzeczywiście posortowana na samym początku - a zatem wykonanie algorytmu jest zbędne. Wszystkie elementy są na właściwych miejscach, a więc liczba sprawdzeń t_i będzie równa 1 przy każdym obrocie zewnętrznej pętli. Wówczas funkcja złożoności przedstawia się następująco:

$$T(n) = 4n - 2 + 2\sum_{i=2}^{n} 1$$

co po uproszczeniu daje nam:

$$T(n) = 6n - 4$$

W najlepszym przypadku złożoność jest więc funkcją liniową względem *n*, czyli jest proporcjonalna do rozmiaru danych.

Taki skrajny przypadek jest bardzo rzadki. Na drugim końcu leży wariant wybitnie pesymistyczny, zakładający maksymalny koszt wykonania algorytmu. W tym przypadku podana tablica jest posortowana, ale… w odwrotnym porządku! Wtedy też wszystkie elementy muszą być kolejno posyłane na początek tablicy: *i*-ty element przemieści się więc o *i* - 1 miejsc do tyłu w każdym obrocie zewnętrznej pętli. Wniosek: $t_i = i$ dla każdego $i = 2, 3, \ldots, n$. Funkcja $T(n)$ będzie zatem wyglądała tak:

$$T(n) = 4n - 2 + 2\sum_{i=2}^{n} i$$

Występująca tu suma nie jest już tak banalna jak w przypadku optymistycznym, Do jej przepisania w bardziej przystępnej postaci najlepiej posłużyć się znanym ci, mam nadzieję, wzorem na sumę najprostszego ciągu arytmetycznego:

$$1 + 2 + \ldots + n = \sum_{k=1}^{n} n = \frac{n(n+1)}{2}$$

A ponieważ my liczymy sumę od 2, finalnie (po kilku przekształceniach, które matematycy określiliby jako „trywialne”) funkcja $T(n)$ przyjmie postać:

$$T(n) = n^2 + 5n$$

W tym przypadku jest to więc funkcja kwadratowa. Czas wykonania algorytmu rośnie więc znacznie szybciej w miarę wzrostu rozmiaru danych.

Tak jest w przypadku najgorszym. Możesz się zdziwić, ale to właśnie tę sytuację powinniśmy przede wszystkim rozpatrywać, gdy mamy na celu określenie efektywności algorytmu! Są ku temu co najmniej trzy powody:

- Przypadek pesymistyczny jest tak zły, że już gorszy być nie może. Wyznaczanie złożoności w tym właśnie przypadku daje nam więc górne ograniczenie na czas działania algorytmu. Innymi słowy, wiemy na pewno, jakiej magicznej granicy czasu wykonania nasza procedura nigdy nie przekroczy. Taka informacja jest

wbrew pozorom znacznie cenniejsza niż średni czas działania: jeśli bowiem przypadek pesymistyczny bardzo odbiega od średniego, to możemy być nieprzyjemnie zaskoczeni, gdy akurat na niego natrafimy. Pół biedy, jeżeli wyłapiemy ten fakt podczas testowania programu. O wiele gorzej, jeżeli spowoduje to irytację końcowego użytkownika, który zlecając programowi rutynową czynność stwierdzi nagle, że wykonuje się ona dwie minuty zamiast czterech sekund. Tłumaczenie się zgubnym wpływem faz Księżyca może wtedy nie być wystarczające…

- Z drugiej strony przypadek pesymistyczny ma tendencję do częstego występowania. Być może niekoniecznie dotyczy to sortowania, ale objawia się regularnie podczas wielu powszechnych operacji, jak np. wyszukiwania. Całkiem prawdopodobna jest przecież próba wyszukania w bazie danych rekordu, który nie istnieje - a to jest właśnie przypadkiem pesymistycznym.
- Wreszcie, wariant malkontencki jest zwykle podobny do przypadku średniego. Weźmy choćby nasze sortowanie: w przypadku średnim liczba przestawień elementów tablicy dokonywanych w *i*-tym obrocie pętli to $t_i = i / 2$. Jeśli masz na to ochotę, możesz zapisać funkcję $T(n)$ dla tego właśnie przypadku; po uproszczeniu zawartej w niej sumy otrzymasz ponownie funkcję kwadratową.

Znajdujemy złożoność teoretyczną

„Zaraz", możesz jednak zaprotestować. „W przypadku średnim będzie to jednak **inna** funkcja kwadratowa, przyjmująca mniejsze wartości dla tych samych *n*!" Nie mogę się z tobą nie zgodzić, bo byłby to zamach na podstawy matematyki. Mogę jednak zadać prowokujące pytanie: A jakie to ma znaczenie?

I spieszę jednocześnie z odpowiedzią, że wcale nie takie duże! Weźmy sobie choćby sytuację, w której sortujemy 1000-elementową tablicę (dość skromny przypadek, nawiasem mówiąc), a więc $n = 1000$. Ponieważ zaś $T(n)$ jest funkcją kwadratową, należy oczekiwać, że liczba elementarnych kroków algorytmu będzie się dla różnych rozkładów danych wahać w okolicach miliona. Czy jednak ma znaczenie dokładna liczba operacji podstawowych? Czy zrobi to jakąś istotną różnicę, gdy algorytm wykona w jednym przypadku milion i pięć tysięcy, a w drugim milion i pięćdziesiąt tysięcy instrukcji?… Nawet nie trzeba liczyć, jakiego rzędu jest to różnica (podpowiedź: to są promile) - możesz empirycznie się przekonać, że dla dzisiejszych komputerów to kwestia mikrosekund.
Naturalnie, można się upierać, że te szczegóły mają znaczenie. Jeśli na przykład współczynnik przy *n* w $T(n)$ wynosiłby kilka setek, to w ostatecznym rozrachunku miałoby to spory wpływ na czas wykonania. Musisz jednak uświadomić sobie, że takie założenia to droga donikąd. Wystarczy wziąć bowiem większe *n* - powiedzmy 10000, znów nie jest to jeszcze bardzo dużo - by wykazać praktyczną identyczność obu pozornie różnych złożoności (średniej i optymistycznej).

Kluczowa cecha algorytmu

Te obserwacje pozwalają nam na uczynienie ostatniego kroku w analizie efektywności algorytmów. Możemy teraz określać ich **złożoność teoretyczną**.

Złożoność teoretyczna (zwana też **klasą algorytmu**) określa, jak silnie zależą od siebie: **rozmiar danych** i **czas wykonania algorytmu** - przy założeniu, że ten pierwszy wzrasta **nieograniczenie**.

Wielkość ta, podana przy opisie konkretnego algorytmu, jest jak CV kandydata o pracę. Patrząc na nią i porównując z innymi rozwiązania dla tego samego zadania możemy łatwiej zdecydować, który algorytm będzie dla nas najodpowiedniejszy. Złożoność teoretyczna jest bowiem **uniwersalną miarą efektywności**.

Notacja

Skoro jest ona tak ważna, powinniśmy nauczyć się nią posługiwać. Na szczęście nie jest to trudne i nie wymaga nawet znajomości matematycznych podstaw, kryjących się za stosowaną notacją.

Wróćmy więc do analizowanego przez cały czas algorytmu `InsertionSort()` i jego funkcji złożoności $T(n)$. Od jakiegoś czas podkreślałem usilnie fakt, że jest to funkcja kwadratowa względem n; jednocześnie przekonywałem, że tak naprawdę nie warto wnikać, jak dokładnie ta funkcja wygląda.
Takie podejście jest właśnie istotą określania złożoności teoretycznej. Bierzemy po prostu pod uwagę ten składnik, który ma w ostatecznym rozrachunku **największy wpływ na wartość** $T(n)$. W naszym przypadku jest to n^2, gdyż to on czyni ją funkcją kwadratową. Przy użyciu standardowego sposobu zapisu wyrażamy to tak, iż: **złożoność algorytmu `InsertionSort()` jest rzędu $\Theta(n^2)$**. Bardziej sformalizowane stwierdzenie to po prostu:

$$T\left(n\right)=\Theta\left(n^2\right)$$

Mówi ono dokładnie to, że funkcja $T(n)$ jest **funkcją kwadratową względem *n***. Wiemy rzecz jasna, że $T(n)$ określa nam także liczbę elementarnych instrukcji wykonywanych podczas sortowania przez wstawianie tablicy n-elementowej. Notacja $\Theta(n^2)$ mówi więc również o tym, iż ilość tych kroków jest **proporcjonalna do kwadratu rozmiaru** sortowanej tablicy.

Asymptotyczność

Jak widzimy, podanie złożoności teoretycznej wskazuje jedynie, jak bardzo czas wykonania algorytmu zależy od rozmiaru danych. Takie uproszczenie jest uzasadnienie z jednego powodu. Zakładamy mianowicie, że tenże rozmiar jest duży - właściwie można by nawet powiedzieć, że dąży do nieskończoności. Z tego powodu zapis $\Theta(n^2)$ (i jeszcze kilka podobnych) określa się jako **notację asymptotyczną**.

Mimo że w toku analizy złożoności przemyciłem już kilka argumentów popierających takie podejście, może ono nadal wydawać ci się nadużyciem. Dlaczego więc mielibyśmy stosować taką nieprecyzyjną specyfikację efektywności?… Powodów jest kilka:

- Założenie, że rozmiar danych dąży do nieskończoności (albo, łagodniej mówiąc, jest bardzo duży) nie jest wcale tak niedorzeczne, jak na pierwszy rzut oka mogłoby się wydawać. Przykład z sortowaniem tysiąca czy dziesięciu tysięcy liczb jest raczej regułą niż wyjątkiem. Podobnie często spotkać się można z wyszukiwaniem w zbiorze danych liczącym miliony rekordów czy odczytywaniem bajtów spośród wielu miliardów zmagazynowanych na dysku twardym. Sytuacja, gdy informacji do przetworzenia jest bardzo dużo nie należy zatem wyjątków. Warto zresztą przypomnieć, że właśnie konieczność obróbki wielkich porcji danych była jedną z przyczyn powstania komputerów…
- Analiza algorytmów celem znalezienia ich złożoności praktycznej najczęściej prowadzi do uzależnienia jej nie tylko od rozmiaru zestawu danych, ale także od innych jego cech. W przypadku sortowania wprowadziliśmy na przykład współczynnik t_i, który w prosty sposób charakteryzował stopień uporządkowania tablicy. Mieliśmy przy tym szczęście, gdyż występował on jedynie w jednym miejscu; ponadto, po przyjęciu założeń co do rozważanego przypadku można było to pojedyncze wystąpienie zredukować jedynie przy pomocy raczej prostych operacji na sumach. W wielu często stosowanych algorytmach nie jest jednak tak różowo. Doprowadzenie funkcji $T(n)$ do sensownej postaci (bez dodatkowych parametrów) może być niekiedy wręcz niemożliwe. Znacznie częściej i łatwiej można natomiast podać asymptotyczne oszacowanie na $T(n)$ - czyli znaleźć złożoność teoretyczną algorytmu.

- Jak będziesz się mógł dowiedzieć, czytając dalej ten rozdział, określenie klasy algorytmu jest częstokroć możliwe wręcz „na oko” - jedynie poprzez uważne przyjrzenie się danemu przepisowi. Wówczas nie tylko nie ma potrzeby dokonywania jakichś skomplikowanych operacji algebraicznych, ale także oznaczania elementarnych instrukcji w procedurze i zapisywania złożoności praktycznej. Łatwość uzyskania klasy algorytmu jest więc kolejnym argumentem przemawiającym za jej stosowaniem.
- I wreszcie ostatni powód, należący do kategorii faktów dokonanych. Otóż złożoność teoretyczna jest powszechnie przyjętym sposobem określania efektywności algorytmach. Oczywiście, w opisach dość często można natrafić na uwagi mówiące o tym, jak dane rozwiązania sprawdza się w praktyce w porównaniu z innymi, o tej samej klasie. Nie zmienia to jednak faktu, iż klasa algorytmu jest najważniejszym czynnikiem determinującym jego rzeczywisty czasy wykonania - ważniejszym od szczegółów technicznych.

Nasze dalsze poszukiwania będą więc koncentrowały się właśnie na tym pojęciu. W następnej sekcji postaram się przedstawić w miarę przystępny sposób tę porcję matematyki, która kryje się za notacją asymptotyczną. Natomiast reszta rozdziału to kilka wskazówek, mających na celu pomoc w znajdowaniu złożoności obliczeniowej algorytmów w typowych sytuacjach (najczęściej bez uciekania się do skomplikowanego aparatu matematycznego).

Notacje asymptotyczne

W tej sekcji powiemy sobie więcej o tym niezbyt oczywistym przy pierwszym kontakcie sposobie wyrażania efektywności algorytmów - czyli o notacjach asymptotycznych.

Mówiłem już wcześniej, że w tym kontekście przymiotnik ‘asymptotyczny’ oznacza, że interesujemy się sytuacjami, gdy rozmiar danych algorytmu rośnie nieograniczenie. Stanowi to matematyczne uogólnienie najczęściej spotykanym w prawdziwym życiu przypadków, gdy zestaw danych dla algorytmu jest faktycznie bardzo duży. Założenie o dążeniu do nieskończonoci pozwala jednak na dokonanie kilku znaczących uproszczeń. Pokazałem pod koniec poprzedniej sekcji, że polegają one na:

- pominięciu wszystkich składników funkcji oprócz tego, który ma największy wpływ na jej wartość (a więc rośnie najszybciej)
- pominięciu wszelkich stałych współczynników

Na tej podstawie mogliśmy więc stwierdzić, że asymptotyczny czas działania algorytmu sortowania przez wstawianie wynosi $\Theta(n^2)$, gdzie n jest wielkością sortowanej tablicy. Teraz wyjaśnimy sobie dokładnie, co ten zapis oznacza, a także wprowadzimy dwie inne, podobne notacje. Omówimy sobie też własności tych notacji, które zdecydowanie ułatwiają określanie klasy algorytmu w praktycznych sytuacjach.

Trzy ważne definicje

Na początku warto uściślić pewien fakt, który niektórych zapewne zbytnio nie wzruszy, ale wielu może co najmniej zdegustować. Otóż notacje asymptotyczne są szeroko stosowane przede wszystkim w informatyce, lecz jako samo pojęcie mają korzenie zdecydowanie matematyczne. Dlatego też przy ich omawianiu należy posługiwać się terminami wziętymi z dziedziny królowej nauk - przede wszystkim pojęciem funkcji.

Algebraiczne pojęcie funkcji jest, mam nadzieję, wszystkim doskonale znane, choć głównie pod postacią funkcji określonych na liczbach rzeczywistych. Nas będą tutaj bardziej interesowały funkcje zdefiniowane dla zbioru **liczb naturalnych** $\mathbb{N} = \{0,1,2,3,4,\ldots\}$. Powód jest oczywisty: parametry określające egzemplarze problemów

dla algorytmów (jak choćby tablice do posortowania) są niemal wyłącznie liczbami naturalnymi.

Teoretycznie nic nie stoi na przeszkodzie, aby podane niżej definicje stosować także dla funkcji liczb rzeczywistych, ale wtedy trudniej o algorytmiczny sens takich pojęć.

W tym paragrafie zdefiniujemy sobie zatem trzy notacje asymptotyczne, używane w odniesieniu do funkcji liczb naturalnych. Jakkolwiek sam fakt takiego matematycznego postawienia sprawy może juz z miejsca być odstręczający, definicje te nie są wcale takie trudne do zrozumienia. W praktyce stosuje się je całkiem intuicyjnie.

Ponieważ w kwestii definiowania notacji asymptotycznych panuje pewien rozgardiasz (wspomnę o nim pod koniec), musiałem zdecydować się na jakiś wybór, który byłby jednocześnie prosty i użyteczny. Posłużyłem się więc definicjami z powszechnie uznanej książki na temat algorytmiki: *Wprowadzeniu do algorytmów* Thomasa H. Cormena i współautorów. One z kolei są wprost bazowane na absolutnej klasyce literatury informatycznej, czyli *Sztuce programowania* Donalda E. Knutha.

Dokładne oszacowanie (notacja Θ)

Zacznijmy od używanego już przez nas symbolu Θ, wykorzystywanego w zapisie $\Theta(n^2)$. Ogólnie jest to notacja postaci $\Theta(f(n))$, co czytamy 'wielkie theta od *f* od *n*'. Z wyglądu zdaje się więc, że mamy do czynienia z „funkcją określoną na innej funkcji" albo z funkcjami zagnieżdżonymi. Naprawdę jest to coś ciekawszego; mamy bowiem do czynienia ze **zbiorem funkcji** określonym mniej więcej tak:

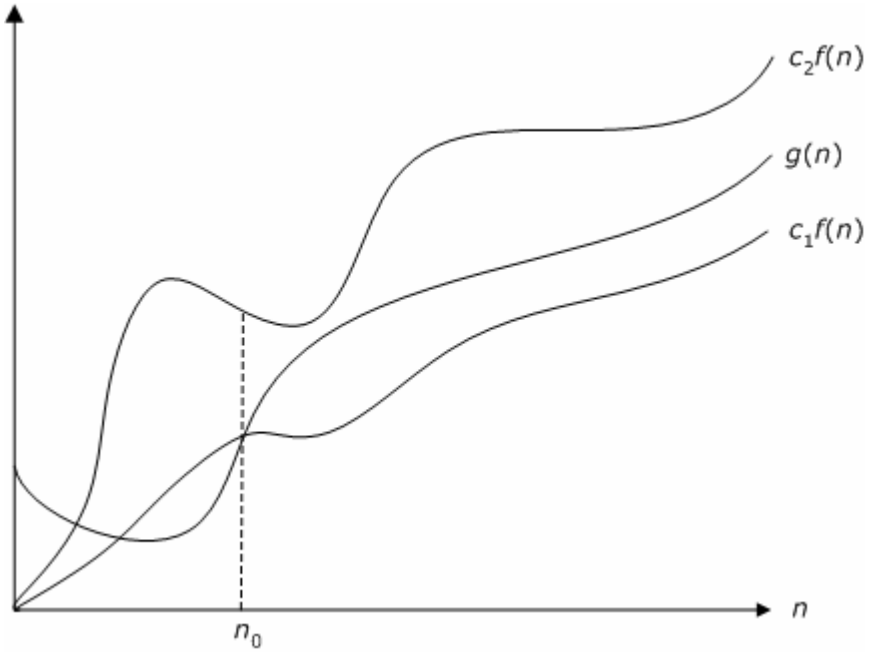
$$\Theta\left(f\left(n\right)\right)=\left\{\begin{array}{l} g\left(n\right):\text{istnieją liczby } c_1,c_2,n_0>0,\text{ takie że} \\ 0\le c_1 f\left(n\right)\le g\left(n\right)\le c_2 f\left(n\right)\text{ dla wszystkich } n\ge n_0 \end{array}\right\}$$

Zbiór funkcji może być dziwnym tworem, jeśli dotąd byłeś przyzwyczajony wyłącznie do zbiorów liczbowych czy punktów na płaszczyźnie albo w przestrzeni. W matematyce elementami zbiorów mogą być jednak obiekty dowolnego rodzaju (nawet inne zbiory), więc nic nie stoi na przeszkodzie, abyśmy połączyli w zbiór $\Theta(f(n))$ funkcje spełniające wyżej wymieniony warunek.

W zwięzłym, dosłownym zapisie matematycznym powyższa definicja wygląda następująco:

$$\Theta\left(f\left(n\right)\right)=\left\{g\left(n\right):\exists c_1,c_2,n_0\in\mathbb{N}^+\ \forall n\ge n_0\quad 0\le c_1 f\left(n\right)\le g\left(n\right)\le c_2 f\left(n\right)\right\}$$

Cóż jednak ten warunek praktycznie oznacza?... Dowolna funkcja $g(n)$ należy do zbioru $\Theta(f(n))$ wyłącznie wtedy, gdy możemy sobie znaleźć trzy wymienione w definicji liczby c_1, c_2, n_0, dla których $g(n)$ jest „wstawiona między" funkcje $c_1f(n)$ i $c_2f(n)$. Ten aspekt najłatwiej zrozumieć, patrząc na wykresy wszystkich trzech funkcji: od pewnego momentu (n_0) funkcja $g(n)$ leży w całości w obrębie „nożyc" tworzonych przez wykresy $c_1f(n)$ i $c_2f(n)$:



Wykres 1. Graficzna interpretacja faktu, iż *g*(*n*) należy do Θ(*f*(*n*))

Oznacza to także, że dla $n \geq n_0$ funkcja $g(n)$ jest równa $f(n)$ z **dokładnością do stałej czynnika** (wartości obu funkcji są do siebie **proporcjonalne**). Mówimy też, że:

Jeśli $g(n)$ należy do $\Theta(f(n))$, to f(n) jest **asymptotycznie dokładnym oszacowaniem** dla $f(n)$.

Zwrot 'dokładne oszacowanie' wydaje się mieć cechy jakiegoś dziwnego paradoksu, jednak ma on sens. Jak się bowiem przekonamy, notacja Θ ma charakter podobny do (względnej) równości obu funkcji - a przynajmniej takiej równości, na którą możemy sobie pozwolić w analizie algorytmów.

Korzystając z podanej definicji możnaby uzasadnić, że znaleziona przez nas funkcja $T(n) = n^2 + 5n$ rzeczywiście jest rzędu $\Theta(n^2)$. Aby uczynić, należy po prostu znaleźć trzy dodatnie stałe c_1, c_2, n_0 tak, aby spełniona była nierówność:

$$c_1 n^2 \leq n^2 + 5n \leq c_2 n^2$$

Nie jest to bardzo skomplikowane - wystarczy podzielić wszystko przez n^2 i rozwiązać obie nierówności przy założeniu, że $n \geq 0$. W typowych przypadkach nie musimy jednak tego robić. W zupełności zadowolić się można nieformalną metodą z poprzedniej sekcji: trzeba po prostu brać pod uwagę tylko **najbardziej znaczacy składnik** i **pominąć wszelkie współczynniki stałe**.

Łatwo można więc zrozumieć, dlaczego każdy wielomian stopnia d jest funkcją rzędu $\Theta(n^d)$ - po prostu bierzemy pod uwagę tylko jego najwyższą potęgę i nie przejmujemy się stałymi współczynnikami. Szczególnym przypadkiem jest $d = 0$; wtedy mamy do czynienia ze stałą - funkcją rzędu $\Theta(n^0)$, czyli po prostu $\Theta(1)$. Taki zapis oznacza więc, że mamy na myśli dowolną liczbę, która jest stała i niezależna od parametru funkcji[2].

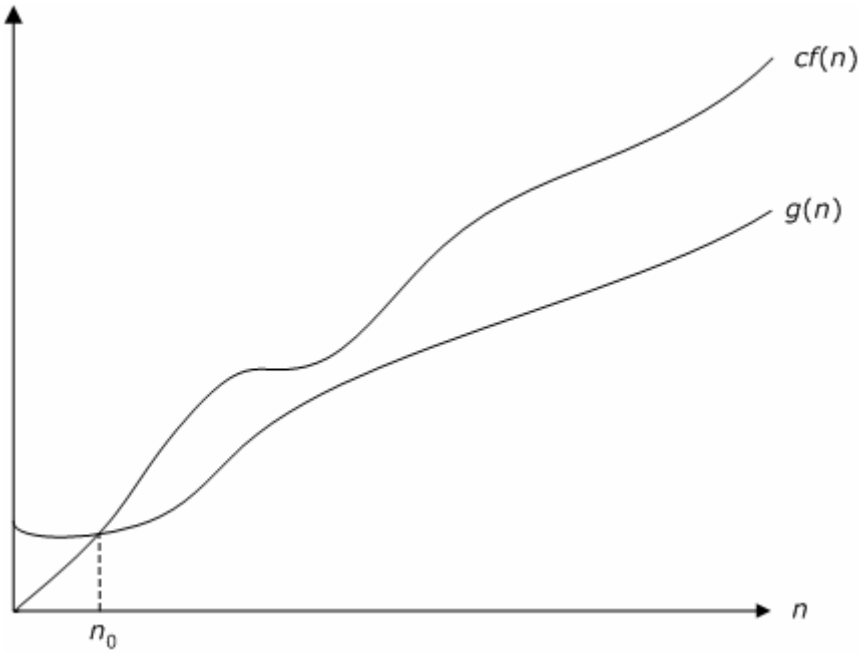
[2] Trochę nieścisłe jst używanie Θ(1), bo wtedy faktycznie nie wiemy, co jest tym parametrem funkcji, od którego Θ(1) jest niezależne. Generalnie jednak użycie tego symbolu w równaniach nie powoduje niejednoznaczności, jeśli wiemy, co w takim równaniu jest zmienną niezależną.

Ograniczenie górne (notacja O)

Druga z notacji jest jakby „górną połową” pierwszej. Napisanie $O(f(n))$, co czytamy jako ‘wielkie O od f o n’[3], oznacza mianowicie zbiór:

$$O\left(f\left(n\right)\right)=\left\{\begin{array}{l} g\left(n\right):\text{istnieją liczby } c,n_0>0,\text{ takie że} \\ 0\le g\left(n\right)\le cf\left(n\right)\text{ dla wszystkich } n\ge n_0 \end{array}\right\}$$

Wszystkie należące do niego funkcje mają więc tę własność, że $f(n)$ ogranicza je od góry. Jeżeli zatem $g(n)$ należy do $O(f(n))$, to przy dążeniu n do nieskończoności wartości $g(n)$ są mniejsze od $f(n)$ co najmniej o stały czynnik (a nawet bardziej). Znowuż najlepiej widać to na wykresie. Mamy mianowicie już tylko jedno „ostrze nożyczek” - to górne, reprezentujące $f(n)$. Od pewnego miejsca (n_0) wykres obrazujący $cg(n)$ cały czas znajduje się więc pod nim. Odpowiada to temu, iż $g(n)$ jest **asymptotycznie mniejsze** od $f(n)$:



Wykres 2. Graficzna interpretacja faktu, że *g*(*n*) należy do *O*(*f*(*n*))

Można też powiedzieć, że $f(n)$ jest **asymptotycznym ograniczeniem górnym** dla $g(n)$. Mając informacje o funkcji wyrażone w postaci notacji O wiemy zatem tylko tyle (albo aż tyle), że funkcja na pewno nie przekroczy tej podanej wraz z zapisem $O(f(n))$ - choćby argument n był maksymalnie duży.

Przekładając to na analizę algorytmów możemy powiedzieć, że znając asymptotyczne ograniczenie górne wiemy, że dany algorytm nie będzie się wykonywał dłużej. Jeśli zaś informacja ta dotyczy przypadku pesymistycznego, mamy całkowitą pewność, że jest to maksymalny czas działania procedury w zależności od rozmiaru danych.
Jak się wkrótce przekonamy, wyznaczenie takiego zgrubnego oszacowania dla algorytmu jest zwykle całkiem proste. W kolejnym podrozdziale zajmiemy się kilkoma technikami dokonywania tego.

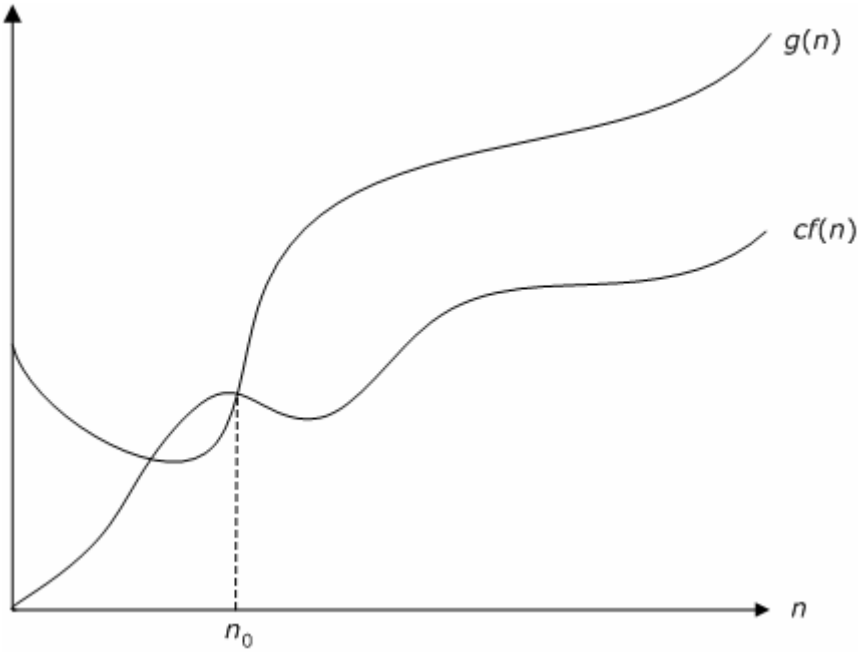
[3] Symbol O nie jest tutaj zwykłą, wielką literą O, lecz grecką literą omikron. Naturalnie, jeśli nie dysponujemy akurat czcionką z greckim alfabetem, posłużenie się łacińskim O nie jest żadnym kardynalnym błędem.

Ograniczenie dolne (notacja Ω)

Jak nietrudno się domyślić, ostatnia notacja jest „dolną połową" pierwszej. Zapis $\Omega(f(n))$ czytamy 'wielkie omega od *f* od *n*', a symbolizuje on ostatni już dzisiaj zbiór funkcji:

$$\Omega\left(f\left(n\right)\right)=\left\{\begin{matrix} g\left(n\right):\text{istnieją liczby } c,n_0>0,\text{ takie że} \\ 0\le cf\left(n\right)\le g\left(n\right)\text{dla wszystkich } n\ge n_0 \end{matrix}\right\}$$

Podobnie jak w przypadku $\mathcal{O}(f(n))$, funkcje należące do $\Omega(f(n))$ są ograniczone tylko z jednej strony. Tutaj jest to limit dolny, a zatem jeśli $g(n)$ należy do $\Omega(f(n))$, to znaczy to, że jest ona **asymptotycznie większa** od $f(n)$. Od pewnego miejsca n_0 wykres $cg(n)$ leży więc w całości ponad wykresem $f(n)$:



Wykres 3. Graficzna interpretacja faktu, iż *g*(*n*) należy do Ω(*f*(*n*))

Łatwo zgadnąć, że mówimy wtedy, iż *f*(*n*) jest **asymptotycznym ograniczeniem dolnym** dla *g*(*n*). Jeżeli więc *f*(*n*) wyraża czas wykonania algorytmu w przypadku optymistycznym, to wiemy na pewno, że lepszych wyników ten algorytm już nie osiągnie - nawet dla najbardziej korzystnych danych.

Obie połówki notacji Θ możemy teraz połączyć w całość. Mianowicie:

Jeżeli $g(n)$ należy zarówno do $\Omega(f(n))$, jak i do $\mathcal{O}(f(n))$, to z pewnością należy także do $\Theta(f(n))$.

Inaczej mówiąc, wyznaczenie identycznych asymptotycznych ograniczeń z góry i z dołu dla danej funkcji pozwala nam na automatyczne ustalenie jej oszacowania. Często w ten właśnie sposób można określić dokładnie klasę algorytmu.

Nieco zamieszania

Jako konstrukcje matematyczne notacje asymptotyczne muszą być ścisłe. Życie wprowadza aczkolwiek nieco bałaganu w ten wyidealizowany obraz. Stosowanie tych notacji wymaga więc jeszcze dwóch uwag na ten temat.

Notacja Θ a O

Jest całkiem prawdopodobne, że z symbolem $O(f(n))$ zetknąłeś się już wcześniej. Jeśli tak, to zapewne był on używany w niezupełnie poprawnym kontekście. Ograniczenie górne jest bowiem często stosowane zamiast dokładnego oszacowania, czyli $\Theta(f(n))$. Takie określanie klasy algorytmu jest mało precyzyjne. Weźmy na przykład nasze sortowanie przez wstawianie: ponieważ możemy powiedzieć o nim, że jest klasy $O(n^2)$, to z pełną powagą można również twierdzić, że należy ono także do klasy $O(n^5)$ czy nawet $O(2^n)$!

Praktyczny sens takich stwierdzeń jest żaden; porównać to można to „precyzyjnego" określania liczby 5 jako „z pewnością mniejszej od miliona", podczas gdy faktycznie nie osiąga nie ona nawet dziesiątki. Oczywiście, nikt poważny nie stosuje notacji O w ten sposób, bo to prowadziłoby jedynie do nieporozumień. Podkreślam jednak, że jest to całkiem **poprawne**, acz nonsensowne.
Dlatego też we wiarygodnych źródłach wiedzy na temat algorytmów bacznie przestrzega się pola zastosowań trzech notacji asymptotycznych. W dalszej części tego rozdziału również będę konsekwentnie rozróżniał zapis $\Theta(f(n))$ i $O(f(n))$, aby nie doprowadzać do błędów.

Równość?…

Jak podkreślałem przy definicjach, wszystkie trzy symbole asymptotyczne są zbiorami - dokładniej mówiąc, zbiorami funkcji. Właściwie więc oznaczenie przynależności do jednego z tych zbiorów powinno się zapisywać np. tak:

$$g(n) \in \Theta(f(n))$$

Oznacza to, że funkcja $g(n)$ należy do zbioru $\Theta(f(n))$ albo że po prostu jest ona tego samego rzędu co $f(n)$. Bardzo często spotyka się jednak zapis w formie równości:

$$g(n) = \Theta(f(n))$$

Pozornie jest on niepoprawny, bo występuje tu swoista „niezgodność typów": po lewej stronie mamy pojedynczą funkcję, a po prawej cały zbiór. Możliwość występowania notacji asymptotycznych w takich równaniach jest jednak bardzo cenna; pozwala ona mianowicie na ukrycie nieistotnych szczegółów. Każde wystąpienie symbolu Θ, O lub Ω w równaniu trzeba po prostu interpretować jako **odpowiednią funkcję** należącą do podanego zbioru. Odpowiednią - to znaczy wybraną tak, aby cała równość była spełniona.

Oczywiście nie zawsze dokładnie wiadomo, o jaką funkcję chodzi. W takich jednak przypadkach jest to zwykle nieistotne. Przykładowo, podanie złożoności algorytmu jako:

$$T(n) = 5n^2 + \Theta(n)$$

znaczy dokładnie tyle, że oprócz składnika $5n^2$ występuje jeszcze jakiś bliżej nieokreślony składnik liniowy. Nie ma on jednakże znaczenia, bowiem funkcja kwadratowa dominuje tutaj zdecydowanie. $T(n)$ jest więc rzedu $\Theta(n^2)$.

Własności notacji asymptotycznych

Podanie kilku definicji, jakkolwiek niezbędne, nie daje nam wszystkiego. Pomówmy więc sobie o kilku ważnych cechach wprowadzonych właśnie notacji asymptotycznych.

Działania na anonimowych funkcjach

Jak zaprezentowałem przed chwilą, notacje te mogą występować w zwyczajnych równaniach. Reprezentuje one wtedy **anonimowe**, bliżej nieokreślone funkcje, które spełniają warunki podane w definicji danej notacji. Przy użyciu anonimowych funkcji często upraszcza się wyrażanie i wyznaczanie złożoności algorytmów.

Aby jednak posługiwać się tymi przydatnymi tworami, powinniśmy poznać ich kilka cech praktycznych. Oto lista paru własności notacji asymptotycznych; występuje na niej wyłącznie notacja Θ, ale wszystkie właśności z powodzeniem odnosza się także do dwóch pozostałych symboli. Spójrzmy więc na to zestawienie:

- $c \cdot \Theta\left(f\left(n\right)\right) \equiv \Theta\left(f\left(n\right)\right)$, gdzie *c* jest stałą.

  Wyjaśnienie tej zaskakującej właściwości pochłaniania czynników stałych jest w gruncie rzeczy proste. Funkcje należące do $\Theta(f(n))$ różnią się od siebie odmienne właśnie dlatego, że ich wartości różnią się o stały czynnik. Czynnik ten może być dowolny, zatem pomnożenie go przez następny stałą w niczym „nie umniejsza jego dowolności". Wynikowy zbiór funkcji jest więc nadal taki sam.
- $\Theta\left(f\left(n\right)\right) + \Theta\left(f\left(n\right)\right) \equiv \Theta\left(f\left(n\right)\right)$

  Ta własność jest również nietypowa, bo zupełnie nie przystaje do znanych nam zasad arytmetyki na liczbach. Można ją jednak wyjaśnić, odwołując się do poprzedniej cechy. $\Theta(f(n)) + \Theta(f(n))$ to inaczej $2\Theta(f(n))$, a ponieważ notacje asymptotyczne pochłaniają czynniki stałe, ostatecznie suma jest równa $\Theta(f(n))$. Identycznie jest dla każdej innej, skończonej liczby składników.
- $\Theta\left(\Theta\left(f\left(n\right)\right)\right) \equiv \Theta\left(f\left(n\right)\right)$

  Jak wspominałem na początku, wystąpienie notacji asymptotycznej należy traktować jako anonimową funkcję. Wewnętrzne $\Theta(f(n))$ reprezentuje więc pewną funkcję; „potraktowanie" jej po raz drugi symbolem Θ daje nam nadal to samo.
- $\Theta\left(f\left(n\right)\right) \cdot \Theta\left(g\left(n\right)\right) \equiv \Theta\left(f\left(n\right) \cdot g\left(n\right)\right)$

  Swoistą rozdzielność względem mnożenia można wytłumaczyć faktem, że notacja asymptotyczna ukrywa w sobie zawsze pewną stałą. Pomnożenie ich przez siebie także daje stałą o identycznym znaczeniu, wobec czego zachodzi powyższa własność.

Oczywiście we wszystkich przypadkach możnaby podać formalne dowody tych i jeszcze kilku innych własności, ale to raczej mija się z celem -w końcu nie jest to podręcznik algebry. Najważniejsze jest, aby móc stosować powyższe cechy notacji w praktyce. W dalszej części rozdziału będziemy się do nich często odwoływać.

Porównywanie funkcji

Dla rozluźnienia po tej pokaźnej dawce formalnej matematyki czas na porcję bardziej intuicyjnego podejścia. Okazuje się, że notacje asymptotyczne wprowadzają coś w rodzaju kryteriów „porównywania funkcji". Relacje te działają podobnie do porównań dwóch liczb rzeczywistych. Tę paralelę ilustruje poniższa tabelka:

relacja między funkcjami	***„odpowiednik" liczbowy***
$g\left(n\right) = O\left(f\left(n\right)\right)$	$f \geq g$
$g\left(n\right) = \Theta\left(f\left(n\right)\right)$	$f = g$
$g\left(n\right) = \Omega\left(f\left(n\right)\right)$	$f \leq g$

Tabela 1. Analogia między notacjami asymptotycznymi a relacjami między liczbami rzeczywistymi

Istnieją też notacje „odpowiadające” niewystępującym to porównaniom $f > g$ i $f < g$. Są one zapisywane jako $g(n) = o(f(n))$ oraz $g(n) = \omega(f(n))$. Ponieważ używa się ich raczej rzadko, nie wprowadzam ich definicji. Zainteresowani mogą rzecz jasna sięgnąć do bardziej fachowej literatury, wymienionej na początku tej sekcji.

Analogia ta ma swoje matematyczne uzasadnienie we własnościach notacji asymptotycznych, jak zwrotność czy przechodniość. Myślę, że nie ma sensu wymieniać ich wszystkich, gdyż o wiele lepiej będzie, jeśli zapamiętasz takie właśnie intuicyjne odwołanie do zwykłych liczb. Gdy tak uczynisz, posługiwanie się tymi notacjami będzie o wiele łatwiejsze.

Uwagi na temat złożoności

Gdy wiemy już, czym jest złożoność obliczeniowa oraz znamy sposoby jej wyrażania, zastanówmy się, co ona właściwie oznacza. W tej sekcji porównamy sobie typowe przykłady klas złożoności algorytmów oraz najbardziej znane rozwiązania, które mają takie właśnie złożoności czasowe. Myślę, że pozwoli to uświadomić sobie, że wprowadzone wcześniej pojęcia nie są jakimś abstrakcyjnym pomysłem natury wyłącznie matematycznej.

Porównanie różnych typów złożoności obliczeniowej

W algorytmice przewija się głównie kilka funkcji, które asymptotycznie określają złożoność obliczeniową wielu algorytmów. Do tych najczęściej spotykanych funkcji należą:

- $\Theta(1)$, gdy czas wykonania algorytmu jest stały i niezależny od rozmiaru danych wejściowych (**złożoność stała**)
- $\Theta(\log n)$, kiedy czas ten rośnie logarytmiczne wraz ze wzrostem wielkości danych (**złożoność logarytmiczna**). Logarytm jest niemal zawsze o podstawie 2 (w przypadku notacji asymptotycznych nie ma to aczkolwiek znaczenia, bowiem podstawa logarytmu może być zmieniona poprzez pomnożenie przez czynnik stały)
- $\Theta(n)$ - czas działania jest proporcjonalny do rozmiaru danych wejściowych (**złożoność liniowa**)
- $\Theta(n \log n)$ - złożoność jest iloczynem funkcji liniowej i logarytmicznej
- $\Theta(n^2)$ - liczba instrukcji algorytmu rośnie proporcjonalnie do kwadratu rozmiaru danych wejściowych (**złożoność kwadratowa**)
- $\Theta(2^n)$ - czas wykonania rośnie wykładniczo względem rozmiaru danych (**złożoność wykładnicza**)
- $\Theta(n!)$ - złożoność jest wyrażona za pomocą silnii (iloczynu wszystkich liczb naturalnych od 1 do n)

Złożoności te zostały uszeregowane według wzrastającego czasu wykonania. Aby uświadomić sobie, jak bardzo klasy te różnią się od siebie, popatrzmy na poniższą tabelkę obrazującą czasy działania algorytmów o różnych złożonościach. Przyjęto w niej, że pojedyncza instrukcja wykonuje się jedną nanosekundę, czyli że algorytm jest wykonywany na komputerze działającym z częstotliwością 1 gigaherca:

***n*→ *T*(*n*) ↓**	***10***	***20***	***50***	***100***	***200***	***1000***
log *n*	3,32 ns	4,23 ns	5,64 ns	6,64 ns	7,64 ns	9,97 ns

n	10 ns	20 ns	50 ns	100 ns	200 ns	1000 μs
n* log *n	33,21 ns	86,44 ns	282,2 ns	664,4 ns	1,53 μs	9,97 μs
n^2	100 ns	400 ns	2,5 μs	10 μs	40 μs	1 ms
2^n	1 μs	1,05 ms	13 dni	$4 \cdot 10^{13}$ lat	$5{,}1 \cdot 10^{43}$ lat	$3{,}4 \cdot 10^{284}$ lat
***n*!**	3,6 ms	77 lat	$9{,}6 \cdot 10^{44}$ lat	$3 \cdot 10^{141}$ lat	$2{,}5 \cdot 10^{358}$ lat	$1.27 \cdot 10^{2551}$ lat

Tabela 2. Przykładowe czasy działania algorytmów o różnych złożonościach dla wybranych rozmiarów danych

Różnice pomiędzy poszczególnymi wyrazami tabeli są, jak widać, gigantyczne. O ile w przypadku czterech pierwszych wierszy wzrost czasu działania algorytmu jest dla człowieka właściwie niezauważalny, o tyle dwie ostatnie funkcje złożoności osiągają wręcz niewyobrażalne wartości.
Dość powiedzieć, że wykonanie algorytmu o złożoności 2^n dla danych o rozmiarze 100 zabierze czas ponadtysiąckrotnie dłuższy od szacowanego wieku Wszechświata! Dla n równego 200 jest z kolei bardzo prawdopodobne, że protony składające się na nasz komputer rozpadną się, zanim zdołamy doczekać się wyniku[4]. Kolejne czasy dla złożoności 2^n, a zwłaszcza $n!$, są nie tylko kwintyliony razy większe niż nawet najbardziej optymistyczne szacunki co do długości dalszego życia kosmosu, ale wręcz nie mają żadnego wyobrażalnego przybliżenia.

Ten zdumiewający rozziew pomiędzy tymi dwoma typami złożoności spowodował, że często mówi się o algorytmach działających w czasie (**pod**)**wielomianowym** oraz **ponadwielomianowym**. W praktyce podział ten jest tożsamy z wyróżnieniem procedur wykonalnych w rozsądnym czasie oraz takich, które z praktycznego punktu widzenia nie zakończą się nigdy. Zdecydowana większość problemów może być na szczęście rozwiązana w czasie wielomianowym lub lepszym. Niemniej istnieje cały szereg zadań, dla których nie są znane tak efektywne rozwiązania; ponieważ wiele z nich ma pewną ciekawą własność, wspomnimy sobie o nich w następnym paragrafie.

Przykłady algorytmów

Teraz masz już pewnie pojęcie, co tak naprawdę kryję się pod poszczególnymi typami złożoności obliczeniowej. Prawdopodobnie jednak nadal zastanawiasz się, jak wielkości odnoszą się do algorytmów faktycznie wykorzystywanych w programach. Spójrz więc na poniższe zestawienie, w którym umieściłem wiele typowych przykładów dla różnych złożoności obliczeniowych:

złożoność	***nazwa algorytmu***	***znaczenie n***	***uwagi***
$\Theta(1)$	instrukcja programu	—	wszystkie pojedyncze instrukcje programów traktuje się tak, jakby ich wykonanie zajmowało stały czas
	operacje na stosach i kolejkach		stosy i kolejki to elementarne struktury danych w programowaniu
$O(1)$	wyszukiwanie w tablicy z haszowaniem	—	haszowanie to specjalny sposób indeksowania elementów tablicy przy pomocy ich wartości
$\Theta(\log n)$	bisekcja	liczba przeszukiwanych danych	algorytm bisekcji służy do wyszukiwania określonego elementu w **posortowanym** zestawie danych
	wyszukiwanie w	liczba węzłów w	drzewo wyszukiwań binarnych

[4] Według fizyków czas życia protonu to 10^{35} lat.

złożoność	***nazwa algorytmu***	***znaczenie n***	***uwagi***
	drzewie BST	drzewie	(BST) jest specjalną strukturą danych, nastawioną na szybkie wstawianie, usuwanie i wyszukiwanie elementów
$O(\log n)$	algorytm Euklidesa	jedna z podanych liczb	słynny algorytm służy do obliczania największego wspólnego dzielnika dwóch podanych liczb całkowitych
$\Theta(n)$	przeszukiwanie liniowe	liczba przeszukiwanych elementów	wyszukiwanie liniowe to po prostu przeglądnięcie całego ciągu **nieposortowanych** danych w poszukiwaniu określonego elementu
	sortowanie przez zliczanie i pozycyjne	liczba sortowanych elementów i/lub ich możliwych wartości	ten rodzaj sortowania nie opiera się na porównywaniu elementów, więc nie jest uniwersalny
	wyszukiwanie wzorca metodą KMP	długość przeszukiwanego tekstu	algorytm Knutha-Morrisa-Pratta jest sposobem na przeszukiwanie tekstów
	operacje na wielomianach (z wyjątkiem mnożenia i dzielenia)	stopień wielomianu	dodawanie, odejmowanie, obliczanie, różniczkowanie i całkowanie wielomianów można wykonywać w czasie liniowym, jeżeli przechowujemy ich współczynniki
$O(n)$	wyznaczanie statystyk pozycyjnych	liczba elementów	statystyka pozycyjna to „miejsce” danego elementu w posortowanym ciągu - tutaj ustalane **bez** wykonywania sortowania
	przeszukiwanie grafu	ilość wierzchołków i/lub krawędzi grafu	przeszukiwanie to przechodzenie po wszystkich wierzchołkach grafu
$\Omega(n \log n)$	algorytm sortowania oparty na porównaniach	liczba sortowanych elementów	żaden algorytm sortowania, które opiera się na porównywaniu elementów, nie może działać lepiej
	znajdowanie najmniejszej otoczki wypukłej	liczba punktów	otoczka wypukła to wielokąt, otaczający całkowicie podany zbiór punktów
$\Theta(n \log n)$	sortowanie przez scalanie (*mergesort*) lub kopcowanie (*heapsort*)	liczba sortowanych elementów	w praktyce lepsze bywa sortowanie szybkie (*quicksort*), choć jego pesymistyczna złożoność to $\Theta(n^2)$
	przemyślne mnożenie wielomianów	stopień wielomianu	algorytm tego mnożenia jest dość skomplikowany, bo wykorzystuje szybką transformatę Fouriera (ang. *Fast Fourier Transform*)
$O(n \log n)$	znajdowanie najmniej odległych punktów	liczba wszystkich punktów	większość czasu tego algorytmu zajmuje posortowanie punktów
$\Theta(n^2)$	sortowanie przez wstawianie	liczba sortowanych elementów	sprawdzają się doskonale dla niewielu elementów

złożoność	***nazwa algorytmu***	***znaczenie n***	***uwagi***
	(*insertion sort*) i bąbelkowe (*bubble sort*)		(kikudziesięciu, kilkuset)
	proste mnożenie wielomianów	stopień wielomianu	odpowiednie dla wielomianów o niewielkich stopniach
$O(n^2)$	znalezienie najkrótszej ścieżki w grafie	liczba wierzchołków i/lub krawędzi grafu	krawędzi w grafie mogą być z wagami lub bez wag
	naiwny algorytm wyszukiwania wzorca	średnia długości wzorca i tekstu	naiwny algorytm przegląda po prostu tekst znak po znaku
$\Theta(n^3)$	naturalny algorytm mnożenia macierzy	rozmiar macierzy kwadratowej	wystarczający w ogromnej większości przypadków
$\Omega(2^n)$	„naiwne" szukanie zmiennych spełniających formułę logiczną	liczba zmiennych w formule	sprawdzanie spełnialności formuły logicznej jest użyteczne np. w optymalizacjach czynionych przez kompilatory
$\Omega(n!)$	„naiwne" rozstrzyganie o istnieniu cyklu Hamiltona	liczba wierzchołków grafu	cykl Hamiltona to sposób na przejście wszystkich wierzchołków grafu, odwiedziwszy każdy dokładnie jeden raz
	„naiwne" rozwiązanie problemu komiwojażera	liczba miast do odwiedzenia	problem komiwojażera polega na wyznaczeniu takiej trasy przejazdu między miastami, która jest nie dłuższa od podanej

Tabela 3. Przykłady algorytmów o różnych złożonościach obliczeniowych

Można zauważyć, że wiele pozornie trudnych problemów daje się rozwiązać w stosunkowo dobrym czasie przy pomocy odpowiedniego algorytmu. Z drugiej strony, sporo zdawałoby się prostych zadań jest obecnie wykonalna jedynie w czasie ponadwielomianowym.

Powyższa tabelka ilustruje też, jak ogromna jest liczba różnych zastosowań dla algorytmów. Z oczywistych względów nie ma tu miejsca na szczegółowe omawianie każdego z tych rozwiązań oraz ich zastosowań. Jeżeli cię to interesuje, powinieneś sięgnąć do literatury poświęconej wyłącznie tym zagadnieniom.

Słówko o NP-zupełności

Podział problemów na rozwiązywalne w czasie wielomianym i ponadwielomianowym odpowiada wyróżnieniu wśród nich tych „łatwych" i „trudnych". Możnaby oczywiście uważać problem o złożoności $\Theta(n^{100})$ za w praktyce trudny, ale prawdopodobnie nie istnieją żadne algorytmy o podobnej charakterystyce. Nawet gdyby były one rozwiązaniami jakichś ważnych problemów, znaleziono by dla nich efektywniejsze odpowiedniki. Jak bowiem wynika z kilkudziesięcioletniego istnienia nauki zwanej algorytmiką, obniżanie złożoności wielomianowej jest nieporównywalnie łatwiejsze od pozbycia się zależności np. wykładniczej.
Niestety, dla wielu problemów nie znamy efektywnych algorytmów, działających w czasie wielomianowym. Spora część z tych problemów ma przy tym wyjątkowo intrygujące właściwości. Dodatkowo komplikują one odpowiedź na pytanie, czy owe efektywne algorytmy istnieją. Myślę, że warto o tym powiedzieć trochę szerzej.

Zacznijmy od zaprezentowania powszechnie stosowanego podziału problemów algorytmicznych na tzw. klasy. Otóż wyróżnia się generalnie dwie takie wielkie klasy:

- klasa P zawiera te problemy, które możemy **rozstrzygnąć** (rozwiązać) w czasie co najwyżej wielomianowym. Zdecydowana większość zagadnień należy do tej właśnie klasy - jak choćby wszystkie zaprezentowane w tabeli z poprzedniego paragrafu, oprócz (najprawdopodobniej) trzech ostatnich
- klasa NP obejmuje te problemy, których rozwiązania moglibyśmy **sprawdzić** w czasie wielomianowym. Prościej mówiąc, jeśli mielibyśmy dane opisujące konkretną sytuację problemową oraz „podarowane" skądś rozwiązanie, to moglibyśmy w efektywny sposób sprawdzić, czy to rozwiązanie jest faktycznie poprawne. Do klasy NP należą wszystkie problemy ze wspomnianej tabeli

Może to być zaskakujące, ale właściwie jedyne, co wiemy na pewno na temat relacji pomiędzy tymi dwoma klasami, jest to, iż P $\subset$ NP. Nietrudno zresztą uzasadnić, dlaczego: jeśli bowiem potrafimy rozwiązać jakiś problem w czasie wielomianowym, tym bardziej potrafimy go sprawdzić w takim czasie.
Od ponad trzydziestu lat otwarta pozostaje natomiast kwestia równości lub nierówności obu tych klas. Jeżeli P = NP, wówczas niemal każdy praktyczny problem byłby do rozwiązania w czasie wielomianowym; prawdopodobnie więc całkiem szybko znajdywano by dlań efektywne algorytmy. Większość informatyków sądzi jednak, że rzeczywistość nie jest taka różowa, a P ≠ NP. Jest ku temu jedna poważna przesłanka…

Jest nią istnienie podklasy problemów nazywanych **NP-zupełnymi** (w skrócie NPC, NPC $\subset$ NP). Ich wyjątkowość zawiera się w dwóch cechach.
Po pierwsze, są to najtrudniejsze problemy w obrębie klasy NP. Oznacza to, że każdy problem NP-zupełny jest przynajmniej tak trudny, jak dowolny inny problem z klasy NP.
Druga właściwość jest znacznie bardziej intrygująca. Otóż udowodniono, że każdy problem z klasy NP może zostać **zredukowany** w czasie wielomianowym w dowolny problem NP-zupełny.
Praktyczna konsekwencja tych faktów jest już być może znana tym, którzy umieją czytać uważnie między wierszami. Wynika z nich mianowicie to, iż kategoryczne orzeczenie w sprawie jednego jedynego problemu NP-zupełnego będzie rzutować na całą olbrzymią klasę NP. Jeżeli znajdziemy wielomianowy algorytm dla jakiegoś problemu NPC, wówczas będzie to oznaczało, że takie algorytmy istnieją dla **każdego problemu z klasy NP**; okaże się więc, że P = NP. Analogicznie, udowodnienie że jakiś problem NP-zupełny nie posiada rozwiązania wielomianowego będzie sygnałem, że P ≠ NP.

Jak już mówiłem, obecnie większość informatyków skłania się ku tezie, że żaden efektywny algorytm dla problemu NP-zupełnego nie istnieje. Argumentują to faktem, iż poszukiwania takich algorytmów były przeprowadzane przez wiele lat na całym świecie i zawsze kończyły się niepowodzeniem. Podobnie jednak było z próbami udowodnienia, że takie algorytmy nie istnieją. W sumie więc to powszechne przekonanie o niemożliwości istnienia wielomianowych rozwiązań dla problemów NPC opiera się raczej na intuicji niż racjonalnych podstawach. Jak zatem rzekł Eistein, zawsze może się znaleźć jakiś „nieuk", który nie wie, że to jest niemożliwe, i… zrobić to.

Jeśli zachęca cię perspektywa rozstrzygnięcia trzydziestoletniego sporu[5], zapewne chciałbyś chociaż zobaczyć przykłady problemów NP-zupełnych. Jest ich całe mnóstwo; przykładowe trzy zajmują ostatnie wiersza tabeli złożoności. Spośród nich szczególnie interesujący jest problem komiwojażera - z oczywistych względów praktycznych.

[5] A przy okazji zgarnięcia okrągłego miliona dolarów. Rozstrzygnięcie związku między P a NP jest bowiem jednym z siedmiu tzw. Milienijnych Problemów, ogłoszonych przez Clay Mathematics Institute w 2000 roku. Na stronie internetowej Instytutu możesz poczytać o szczegółach problemu „P vs NP".

W poszukiwaniu złożoności obliczeniowej

W drugim podrozdziale zajmiemy się znajdowaniem złożoności obliczeniowej algorytmów na konkretnych przykładach. Zobaczymy więc, jak poszczególne elementy algorytmów wpływają na ich efektywność oraz jak należy łączyć te wyniki w całość.

Podstawowe zasady

Zaczniemy właśnie od tego łączenia. Musimy bowiem poznać dwie zasady, które umożliwią nam określenie złożoności całego algorytmu w sytuacji, gdy znamy te dane jego poszczególnych „kawałków".

Prawo dodawania

Najbardziej typową sytuacją w programowaniu jest występowanie po sobie kilku instrukcji. W ogólnym przypadku chodzi nam o fragmenty kodu, z których każdy ma swoją określoną złożoność. Oto przykład:

```
int nZmienna = 5;     // Θ(1)
nZmienna += 4;        // Θ(1)
```

Pytanie naturalnie brzmi: Jaką złożoność ma podany fragment jako całość? Nic prostszego - wystarczy **dodać złożoności cząstkowe**:

$$\Theta(1)+\Theta(1)=2\Theta(1)=\Theta(1)$$

Reguły dodawania notacji asymptotycznych mogą nadal wydawać ci się dziwne, ale przecież powyższy wynik można łatwo uzasadnić intuicyjnie. Pojedyncza instrukcja wykonuje się w czasie stałym, zatem stała liczba takich instrukcji także będzie wykonywać się w czasie stałym.
Podobnie byłoby, gdyby instrukcja nie była elementarnym krokiem, lecz np. wywołaniem funkcji o jakiejś złożoności:

```
FunkcjaA();     // Θ(n)
FunkcjaB();     // Θ(n)
```

Na mocy reguł dodawania złożoność powyższego kawałka kodu jest więc rzędu $\Theta(n)$.

Ciekawiej sprawa wygląda, gdy instrukcje mają różne złożoności - jak na przykład tutaj:

```
FunkcjaC();     // Θ(n)
FunkcjaD();     // Θ(n^2)
```

Nadal jednak możemy tutaj korzystać z zasad dodawania notacji - jeżeli oczywiście będziemy pamiętać o kilku innych własnościach. Zobaczmy więc:

$$\Theta(n)+\Theta(n^2)=\Theta(n+n^2)=\Theta(n^2)$$

Najprościej sformułować tutaj zasadę, iż „**silniejszy wygrywa**". Największa złożoność w danym fragmencie kodu dominuje w nim - podobnie jak największy składnik funkcji decyduje o jej asymptotycznej złożoności (jak w równaniu powyżej). Możemy więc mówić, iż:

Algorytm ma taką złożoność, jak jego **najbardziej czasochłonny fragment**.

W dalszym ciągu podrozdziału zobaczysz wręcz, że wyznaczenie złożoności całego algorytmu bardzo często będzie ograniczało się jedynie do określenia jej dla najbardziej czasochłonnego elementu. Złożoność pozostałych fragmentów kodu będzie miała bowiem nikłe znaczenie.

Prawo mnożenia

Druga zasada jest stosowana w nieco innej sytuacji. Przypuśćmy, że znamy złożoność jakiegoś fragmentu kodu i wiemy też, jak często (w funkcji rozmiaru danych) będzie się on wykonywał. Takie przypadku występują w pętlach oraz przy wykorzystaniu rekurencji. Oto przykład:

```
// (wiemy, że poniższa instrukcja wykonuje się Θ(n) razy)
FunkcjaE();    // Θ(log n)
```

Mamy tu więc funkcję o złożoności $\Theta(\log n)$, o której wiemy, że wykona się $\Theta(n)$ razy (bo np. jej wywołanie jest wewnątrz pętli wykonującej *n* cykli). Jaka będzie złożoność całej takiej sekwencji?… Jak wskazuje na to nazwa paragrafu, obie wielkości należy **pomnożyć**:

$$\Theta(n)\cdot\Theta(\log n)=\Theta(n\log n)$$

Jest to zresztą zgodne z intuicją - to samo zrobilibyśmy, działając na liczbach. Pamiętając rzecz jasna o wprowadzonych w poprzednim podrozdziale zasadach mnożenia notacji asymptotycznych, otrzymujemy ostateczenie wynik $\Theta(n\log n)$.

Zasada mnożenia jest szczególnie ważna i często stosowana w przypadku pętli. Generalnie jednak można ją wykorzystywać w każdym przypadku, gdy instrukcja jest w algorytmie wykonywana określoną ilość razy.

Pętle

Pętle są jednym z głównych elementów konstrukcyjnych dla procedur. Ogromna większość algorytmów opiera się na jednej lub kilku pętlach - występujących po sobie i/lub zagnieżdżonych. Należałoby zatem wiedzieć, jak w prosty sposób określić złożoność takich konstrukcji. Na całe szczęście taki prosty sposób istnieje.

Ilość cykli w pętli

Pierwszą rzeczą, jaką trzeba zrobić, jest określenie ilości cykli pętli jako funkcji rozmiaru danych dla algorytmu. A mówiąc prościej: należy ustalić, ile razy dana pętla się wykonuje i wyrazić tę wartość w zależności od *n*. Często jest to zadaniem bardzo prostym; przykładowo, jeśli w naszym algorytmie rozmiarem danych jest liczba elementów tablicy, to naturalne jest, iż pętla:

```
for (unsigned i = 0; i < aTablica.length(); ++i)
   // ...
```

wykona się właśnie *n* razy, bo tyle jest elementów tablicy. Zwróćmy też uwagę, że dokładna ilość cykli nie jest potrzebna, bo, jak wiemy, interesuje nas tylko wielkość asymptotyczna. Nie ma więc znaczenie, czy licznik inicjujemy na `0`, `1` czy `2` - pętla i tak wykona się $O(n)$ razy. Prawdopodobnie ma ona zatem złożoność liniową (co jeszcze sprawdzimy w przyszłym paragrafie).

Niekiedy nie można dokładnie określić liczby cykli. Tak było choćby w przykładzie z algorytmem sortowania przez wstawianie. Wiemy jednak, co trzeba zrobić w takiej sytuacji. Musimy wykazać się pesymistycznym podejściem do życia i założyć, że liczba obrotów pętli będzie **największa**. W większości przypadków będzie to (asymptotyczną) prawdą, lecz uzyskane w ten sposób ograniczenie $\mathcal{O}(n)$ jest **zawsze** prawdziwe.

Złożoność

Wiedząc to, jesteśmy już tylko o krok od określenia złożoności dla dowolnych pętli - a przynajmniej tych, których liczba cykli jest proporcjonalna do *n*. W tym celu musimy jeszcze wiedzieć, ile wysiłku zajmuje wykonanie pojedynczego cyklu; innymi słowy: co kryje się pod wykomentowanym wielokropkiem w przykładzie z poprzedniego paragrafu?…
W ogólnym przypadku tego nie wiemy, więc złożoność jednego cyklu oznaczymy sobie po prostu jako $\mathcal{O}(f(n))$. Jeżeli natomiast tych cykli jest w sumie $\mathcal{O}(n)$ (co ustaliśmy kilka chwil temu), to na mocy prawa mnożenia złożoność całej pętli wynosi:

$$O(n)\cdot O\left(f(n)\right)=O\left(n\cdot f(n)\right)$$

Taki ogólny rezultat jest pożyteczny, ale warto przyjrzeć się też bardziej wyspecjalizowanym.

Jeśli $f(n) = \mathcal{O}(1)$, to jeden cykl pętli zajmuje czas stały. To bardzo typowa sytuacja - występuje chociażby w algorytmie wyszukiwania liniowego, gdy po kolei przeglądamy wszystkie elementy tablicy. Naturalnie, złożoność pętli jest wtedy rzędu $\mathcal{O}(n)$.

Najbardziej interesujący jest jednak przypadek, gdy pętle są **zagnieżdżone**. Oto całkiem typowy przykład:

```
for (unsigned i = 0; i < aTablica.length(); ++i)
   for (unsigned j = i; j < aTablica.length(); ++j)
         // ... (ale wiemy, że to O(1))
```

Idąc „od środka” możemy określić złożoność wewnętrznej pętli jako $\mathcal{O}(n)$: cykl o stałym czasie jest bowiem wykonywany dla (w najgorszym przypadku) wszystkich elementów tablicy wielkości *n*. Jednocześnie to $\mathcal{O}(n)$ jest czasem wykonania cyklu dla zewnętrznej pętli; ona również wykonuje $\mathcal{O}(n)$ obrotów. W sumie więc mamy $\mathcal{O}(n)$ cykli po $\mathcal{O}(n)$ cykli po $\mathcal{O}(1)$ instrukcji, co daje w rezultacie złożoność kwadratową:

$$O(n)\cdot O(n)\cdot O(1)=O(n\cdot n)=O\left(n^2\right)$$

Wreszcie, ogólnijmy wynik na przypadek dowolnego zagnieżdżenia pętli[6]:

```
for (/* ... */)                  // O(n)
   for (/* ... */)               // O(n)
         for (/* ... */)         // O(n)
         // ...                  // itd.
                              // (pojedynczy cykl o złożoności O(1))
```

Gdy więc mamy więc mamy *k* poziomów zagnieżdżenia, to złożoność tego potworka będzie wyrażała się mniej więcej tak:

[6] Wszędzie używam pętli `for`, ale rzecz jasna wszystko dotyczy dowolnych pętli. Mam nadzieję, że było to oczywiste od samego początku…

$$\underbrace{O(n)\cdot O(n)\cdot\ldots\cdot O(n)}_{k \text{ czynników}}\cdot O(1) = O\left(\prod_{i=1}^{k} n\right) = O\left(n^k\right)$$

I to jest w zasadzie najważniejszy wniosek z całej tej zabawy. Widząc (zagnieżdżoną) pętlę będziesz mógł teraz szybko określić jej złożoność. I tak dla pojedynczej iteracji k = 1, więc klasą jest $\mathcal{O}(n)$; dla dwóch poziomów jest to $\mathcal{O}(n^2)$, i tak dalej. Zapamiętajmy więc, że:

***k*-krotnie zagnieżdżona pętla**, której „najbardziej wewnętrzny" cykl wykonuje się w **czasie stałym**, ma złożoność ograniczoną z góry przez $\mathcal{O}(n^k)$.

Nie można oczywiście stosować tego prawa bezmyślnie do każdej pętli. Jeśli dana iteracja nie wykonuje się w czasie proporcjonalnym do liczby elementów (choćby w przypadku pesymistycznym), wówczas nie należy korzystać z tego twierdzenia. Takie przypadki są aczkolwiek niezbyt częste.

Rekurencja

Oto drugie ważne narzędzie w rękach projektanta algorytmów. Tytułowa **rekurencja** (ang, *recurrency*), bo o niej mowa, zwana jest też czasem rekursją albo nieco myląco - wywołaniem zagnieżdżonym. Ogólnie rzecz ujmując jest to sytuacja, gdy dana procedura w określonych okolicznościach **wywołuje samą siebie**. To zastrzeżenie jest ważne, aby rekurencja była poprawna - to znaczy nie prowadziła do nieskończonego ciągu wywołań funkcji. Sytuacja taka jest w pewnym stopniu podobna do nieskończonej pętli, tyle że łatwiej wykrywalna. „Kręcąca" się w nieskończoność pętla wizualnie zawiesza program, natomiast niewłaściwa rekurencja powoduje **przepełnienie stosu** (ang. *stack overflow*). W zależności od stosowanego języka czy platformy sprzetowej powoduje to restart systemu, awaryjne zakończenie programu albo wyjątek czasu wykonania.

Zdania co do przydatności rekurencji w konstruowaniu algorytmów są wysoce podzielone. Można spotkać się z twierdzeniem, że jest to wręcz naturalna metoda ich tworzenia; z drugiej strony wiele jej bardziej skomplikowanych zastosowań jest bardzo wymyślnych i wcale nie oczywistych.
Ponieważ jednak mamy się skoncentrować na analizie efektywności algorytmów rekurencyjnych, zostawmy kwestie jakkolwiek rozumianej „słuszności" czy „naturalności" użycia rekurencji. W tej sekcji spotkasz więc zarówno wiele procedur, które ewidentnie proszą się o odpowiedniki iteracyjne (będzie tak zwłaszcza na początku), jak i nieco bardziej wyrafinowane przypadki - szczególnie związane z techniką zwaną „dziel i zwyciężaj". W każdym przypadku będziemy jednak zainteresowani głównie klasą danego algorytmu i sposobem na jej proste i szybkie znalezienie.

Ogólne metody

Na początek zajmiemy się najprostszymi (i trochę sztucznymi) przykładami użycia rekurencji. Dwie opisane tu techniki mogą jednakże pomóc w znalezieniu złożoności wielu rzeczywistych algorytmów - a przynajmniej dostarczyć w tym zadaniu pewnych wskazówek.

Rozpisywanie

Pisząc ten akapit miałem spore problemy z wyborem algorytmu, który stosowałby rekurencję w odpowiednio prosty sposób, a jednocześnie nie narzucał od razu równoważnego (i najczęściej lepszego) rozwiązania z użyciem pętli. Moje poszukiwania nie zostały niestety uwieńczone sukcesem i mogę z dużą dozą prawdopodobieństwa stwierdzić, że każdy możliwy tutaj przykład byłby równie naciągany. Celem przebrnięcia

przez ten akapit musisz więc to zignorować i potraktować po prostu jako rozgrzewkę przed bardziej uzasadnionymi przypadkami rekurencji.

Najprostsza rekurencja

Przykładem będzie znowu przeszukiwanie jednowymiarowej tablicy. Jest to czynność tak powszechna, znana i prosta, że z pewnością każdy początkujący programista zaznajomiony z konstrukcją pętli zakodowałby z łatwością (albo zajrzał na początek tego rozdziału). Ażeby więc wywrócić do góry nogami tę oczywistość, zaproponujemy rozwiązanie rekurencyjne. Dla danej tablicy wygląda ono tak:

- weź jej pierwszy niesprawdzony element i porównaj z szukanym. Jeśli porównanie się powiedzie (wartości są równe), zwróć indeks znalezionego elementu i zakończ procedurę
- gdy natomiast test okaże się nietrafiony, zastosuj identyczną procedurę dla tablicy złożonej ze wszystkich elementów tuż za tym przed chwilą sprawdzonym
- jeśli przeszukiwana tablica jest pusta, zwróć informację o nieznalezieniu elementu

Opis ten przekłada się prosto na kod C++, realizujący wyszukiwanie określonej liczby w tablicy `int[]`:

```
int Szukaj(const int* pTablica, int nSzukany, unsigned uRozmiar,
unsigned uIndeks = 0)
{
   // jeśli dotarliśmy do końca tablicy, zwróć -1
   if (uIndeks >= uRozmiar)      return -1;

   // sprawdź, czy pierwszy niesprawdzony element jest równy szukanemu
   if (pTablica[uIndeks] == nSzukany)
         return uIndeks;
   else
         // jeśli nie jest, wywołaj rekurencyjnie procedurę
         // dla tablicy pomniejszonej o tej element
         return Szukaj(pTablica, nSzukany, uRozmiar, uIndeks + 1);
}
```

Jeśli wcześniej nie miałes zbyt intensywnego kontaktu z rekurencją, może on wydawać się nieco dezorientujący. Łatwo jednak sprawdzić, że działa on identycznie z wersją iteracyjną. Widać nawet zupełnie oczywiste analogie, jak np. parametr `uIndeks` jako odpowiednik licznika pętli. Można też zauważyć, że program ten jest pisany niejako „od tyłu" w tym sensie, że najpierw umieszczamy kod sprawdzający przypadek specjalny - brak elementu. Teraz jest to jednak warunek przerwania rekurencji (tzw. **warunek terminalny**), który musi zostać sprawdzony, zanim zrobimy cokolwiek innego. W każdej procedurze rekurencyjnej jest to część nieodzowna!

Analiza

Naturalne pytanie brzmi teraz: co z pesymistyczną złożonością powyższego algorytmu? W przypadku wersji iteracyjnej można bardzo łatwo (stosując wskazówki z poprzedniej sekcji o pętlach) wyznaczyć ją na $\Theta(n)$. A jak jest tutaj?… Aby się o tym przekonać, zastosujemy „tradycyjne" rozwiązanie, czyli znajdziemy funkcję $T(n)$.

Rekurencja w naszej procedurze polega na wywoływaniu jej dla coraz mniejszych tablic: z każdym jej poziomem przeglądana tablica jest mniejsza o jeden element. Funkcja $T(n)$ będzie więc **także rekurencyjna**: jej wartość dla n będzie zależna od wartości n-1. W jaki sposób?
Spójrzmy na ciało procedury. W każdym poziomie rekurencji, oprócz kolejnego wywołania rekursywnego, dokonywane są jeszcze dwa porównania: sprawdzenie rozmiaru tablicy i aktualnego elementu. Ponieważ wspomniane wywołanie pracuje już na tablicy mniejszej o jeden element, funkcja $T(n)$ będzie się więc na razie przedstawiać następująco:

$$T(n) = T(n-1) + 2$$

Nie możemy jednak zapomnieć o warunku terminalnym. U nas zachodzi on wtedy, gdy algorytm, że podana mu tablica jest pusta. Rozważamy przypadek pesymistyczny (brak szukanego elementu), zatem taka sytuacja z pewnością zajdzie. Odpowiada ona sytuacji, gdy $n = 0$; wówczas procedura dokonuje jednego porównania i natychmiast kończy się. Ostatecznie więc funkcja złożoności z uwzględnieniem koniecznego warunku wygląda w ten sposób:

$$T(n) = \begin{cases} 1 & \text{dla } n = 0 \\ T(n\text{-}1) + 2 & \text{dla } n \geq 1 \end{cases}$$

Jest ona rekurencyjna, zatem nie pozwala na bezpośrednie określenie klasy algorytmu. Musimy więc doprowadzić ją do sensowniejszej postaci.

Rozwiązanie rekurencji

Prostym sposobem, który niestety działa raczej rzadko, jest rozpisanie powyższej funkcji od wartości *n* aż do zera:

$$\begin{gathered} T(n) = T(n-1) + 2 \\ T(n-1) = T(n-2) + 2 \\ T(n-2) = T(n-3) + 2 \\ T(n-3) = T(n-4) + 2 \\ \vdots \\ T(1) = T(0) + 2 \\ T(0) = 1 \end{gathered}$$

Dodając stronami te równania otrzymamy jedno bardzo rozbudowane, które na szczęście będziemy mogli zaraz uprościć:

$$T(n) + T(n-1) + \ldots + T(0) = \left(T(n-1) + 2\right) + \left(T(n-2) + 2\right) \ldots + 1$$

Należy w tym celu zauważyć, że:

- po prawej stronie mamy *n* składników zawierających dwójki, więc dodanie tych dwójek da nam wyniku po prostu $2n$
- składniki od $T(n\text{-}1)$ do $T(0)$ występują w sumie po obu stronach równania. Wszystkie więc mogą być natychmiast zredukowane

W wyniku tych operacji po lewej stronie zostaje nam jedynie $T(n)$, zaś po prawej - nierekurencyjna postać tej funkcji:

$$T(n) = 2n + 1$$

Teraz już rzecz jasna nie ma żadnych kłopotów z określeniem klasy algorytmu. Jak można było się spodziewać i co wykazaliśmy przed chwilą, jest ona również rzędu $\Theta(n)$. Postać rekurencyjna wyszukiwania wydaje się więc nie różnić od wersji iteracyjnej[7].

[7] W praktyce jest inaczej. Wersja rekurencyjna wymaga dużo dodatkowej pamięci dla stosu, co ostatecznie bardzo spowalnia jej wykonywanie. Istnieje też ryzyko przepełnienia stosu, co naturalnie nie występuje w

Drzewo rekursji

Metoda polegająca na rozpisaniu i dodaniu do siebie stronami ciągu równań jest prosta i prowadzi do dokładnego rozwiązania (czyli wyznaczenia nierekurencyjnej postaci funkcji). Jak już jednak wspomniałem, można ją stosować rzadko, właściwie tylko w wyjątkowych sytuacjach. Gdyby np. zamiast $T(n\text{-}1)$ umieścić w funkcji $2T(n\text{-}1)$, sposób ten doprowadziłby raczej do jeszcze większego skomplikowania całej sprawy.

Bardziej efektywna i mająca szersze pole zastosowań metoda nie zapewnia dokładnego rozwiązania, lecz przecież dla znalezienia klasy algorytmu nie jest ono potrzebne. Metodę tę zaprezentuję na ponownie banalnym przykładzie.

Słynny ciąg

Jednym z bardziej znanych dziwolągów matematycznych (rozsławionym przez autorów powieści sensacyjnych ze względu na swoje kryptograficzne właściwości) jest **ciąg Fibonacciego**. Jego cechą szczególną jest to, że każdy wyraz powstaje przez zsumowanie dwóch poprzednich:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...

Twór ten ma też mnóstwo innych cech, jak np. zdolność do opisywania wzrostu populacji królików oraz fraktalne rozmieszczenie zer i jedynek w binarnej reprezentacji swoich wyrazów. Dla nas ciąg ten będzie ważny ze względu na rekurencyjny algorytm obliczania jego n-tego wyrazu. Bezpośrednio z faktu, że jest on sumą dwóch go poprzedzających (oraz tego, że dwa początkowe wyrazy ciągu to jedynki), wynika taki oto wzór:

$$F_n = \begin{cases} 1 & \text{dla } n \in \{1,2\} \\ F_{n-1} + F_{n-2} & \text{dla } n > 2 \end{cases}$$

Możesz powiedzieć, że nazywanie go ‘algorytmem’ jest lekką przesadą. Tym niemniej przekłada się on na odpowiednią funkcję:

```
unsigned Fib(unsigned n)
{
   if (n <= 2) return 1;
   else return Fib(n - 1) + Fib(n - 2);
}
```

Jak widać, występują tu dwa wywołania rekurencyjne, obliczające dwa wyrazy ciągu poprzedzające ten żądany. Wynikają one rzecz jasna wprost z wzoru zaprezentowanego wyżej.

Analiza

Niniejszy algorytm (mimo wszystko pozostanę przy tym określeniu…) jest krótki i prosty. Intuicyjnie wydawałoby się więc, że jego złożoność jest niewielka. Prawda okaże się co najmniej zaskakująca… No, ale nie uprzedzajmy faktów.

Zacznijmy od zapisania złożoności praktycznej $T(n)$. Dla $n = 1$ lub 2 funkcja dokonuje tylko jednego porównania, po czym od razu zwraca wynik; są to terminalne przypadki dla rekurencji. W innych przypadkach oprócz rzeczonego porównania następują też dwa dalsze wywołania rekurencyjne. W sumie więc funkcja $T(n)$ wyglądać będzie tak:

---

przypadku pętli. Widać zatem, że w tym przypadku wersja algorytmu z wykorzystaniem pętli jest o wiele lepsza.

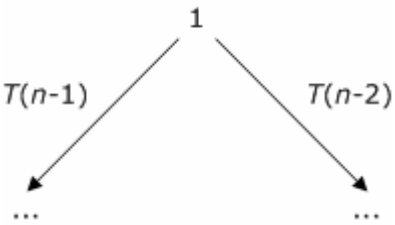
$$T(n)=\begin{cases}1 & \text{dla } n\le 2\\ T(n-1)+T(n-2)+1 & \text{dla } n>2\end{cases}$$

Jak można się było spodziewać, jest to znowu funkcja rekurencyjna. Aby określić jej klasę musimy zatem spróbować doprowadzić ją do bardziej przejrzystej postaci.

Rozwiązanie rekurencji

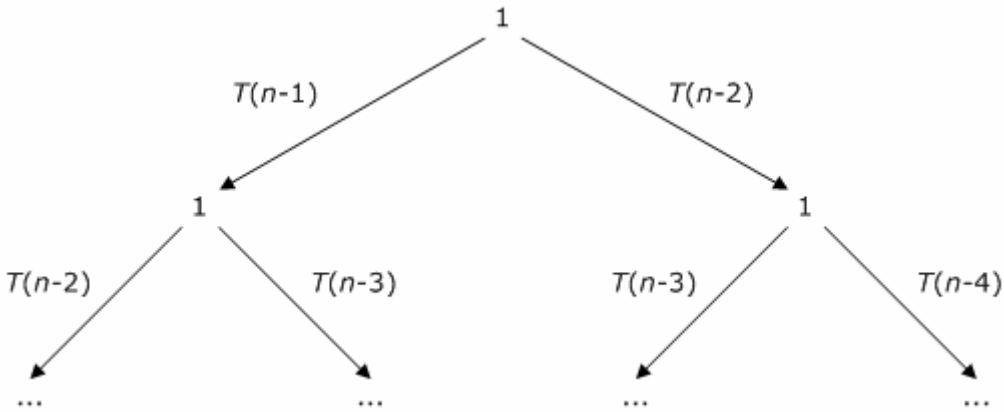
Zastosowanie metody rozpisywania, choć wydaje się logiczne, nie doprowadzi niestety do rozwiązania. Nie ma bowiem szans na takie zredukowanie równania, aby pozostał jedynie *T*(*n*) i składniki nierekurencyjne. W tym przypadku trzeba wykazać się nieco innym podejściem do sprawy.

Pierwsza obserwacja polega na zauważeniu, że wartość *T*(*n*) składa się z trzech składników, z których jeden jest stały, czyli znany nam, natomiast pozostałe dwa są wywołaniami rekurencyjnymi **o takich samych właściwościach**. Oznacza to, że one również składają się z trzech składników, z których dwa są rekurencyjne, itd.
Pomysł polega na zbudowaniu **drzewka**, którego węzły reprezentowałyby znane znam, stałe składniki, zaś krawędzie (gałęzie) - wywołania rekurencyjne. Na początku takie drzewo wyglądałoby dość skromnie:



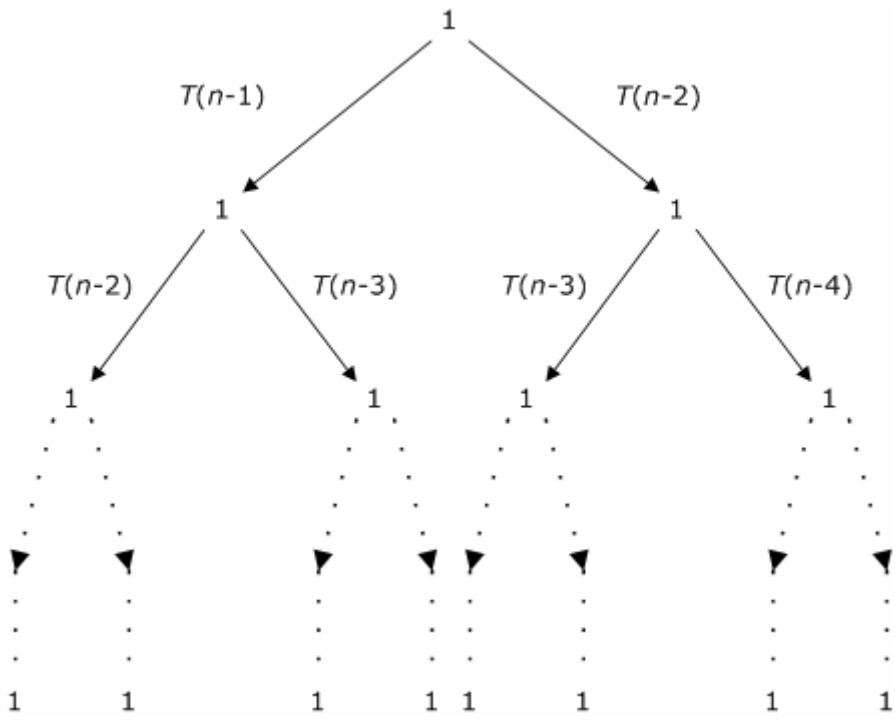
Schemat 1. Drzewo rekursji dla złożoności praktycznej ciągu Fibonacciego - etap pierwszy

Nietrudno je jednak rozbudować. Ukryte za wielokropkami poddrzewa tworzymy bowiem… rekurencyjnie - z tym, że teraz *T*(*n*-1) lub *T*(*n*-2) pełnią rolę *T*(*n*):



Schemat 2. Drzewo rekursji dla złożoności praktycznej ciągu Fibonacciego - etap drugi

W podobny sposób budujemy drzewko, aż dojdziemy do liści, czyli węzłów na końcach wywołań *T*(2) lub *T*(1). Po zsumowaniu wartości we wszystkich węzłach drzewa otrzymamy liczbę kroków algorytmu potrzebnych do obliczenia *n*-tej liczby Fibonacciego. Stąd już bardzo blisko do określenia klasy tego algorytmu.



Schemat 3. Gotowe drzewo rekursji dla złożoności praktycznej ciągu Fibonacciego

W naszym przypadku w węzłach mamy wyłącznie jedynki, zatem problem sprowadza się do określenia ich ilości. W tym celu wyobraźmy sobie, jak wygląda drzewko dla wartości $T(3)$ - jest to pierwsza wartość, dla której występują składniki rekurencyjne. Drzewo będzie miało wyłącznie jedno rozgałęzienie, wysokość 1 i dwa poziomy. Generalnie można stwierdzić, że dla danego n drzewo $T(n)$ ma wysokość n-2 oraz n-1 poziomów. A ile jest węzłów na każdym poziomie?… Na pierwszym mamy oczywiście jeden korzeń; na drugim są już dwa węzły, odpowiadające dwóm wywołaniom rekurencyjnym. Od każdego z nich odchodzą po dwie krawędzie, zatem na trzecim poziomie mamy cztery węzły, i tak dalej - schodząc niżej, (zazwyczaj) podwajamy ilość węzłów. A ponieważ liczba poziomów drzewa wynosi n-1, na ostatnim poziomie będzie więc mniej więcej 2^{n-1} liści, zaś na i-tym - około 2^{i-1} węzłów.

Nie możemy dokładnie określić tych wartości, gdyż drzewo nie zawsze jest zrównoważone. Z grubsza rzecz biorąc znaczy to, że najniższy poziom nie zawsze zawiera wszystkie liście drzewa. Przykładem może być np. drzewko dla $T(4)$. Liczba liści różni się jednak co najwyżej o stałą, zatem nie wpływa to na złożoność asymptotyczną.

By uzyskać przybliżoną ilość węzłów w drzewie należy oczywiście zsumować ich ilości na wszystkich n-1 poziomach:

$$1+2+4+8+\ldots+2^{n-1}=\sum_{i=0}^{n-1}2^i=\sum_{i=0}^{n}2^i-2^n=2^{n+1}-2^n=2^n$$

Otrzymujemy funkcję wykładniczą względem n. Możemy zatem stwierdzić, że złożoność rekurencyjnego algorytmu dla ciągu Fibonacciego to aż $\mathcal{O}(2^n)$.

Post scriptum: algorytm iteracyjny dla ciągu Fibonacciego

Rekurencyjna procedura obliczania ciągu jest więc skrajnie nieefektywna. W sumie nie jest to powód do jakiegoś szczególnego zmartwienia, bo jest spora szansa, że ów ciąg nie będzie ci nigdy do niczego potrzebny. Jeśli jednak zdarzy się inaczej, to zdecydowanie powinieneś wtedy poszukać innego rozwiązania.

Problem z rekurencją polega tutaj na tym, iż większość wyrazów jest obliczana wielokrotnie, przez co mnóstwo czasu procesora po prostu się marnuje. Procedura ta jest po prostu mało inteligentna. W takich wypadkach stosuje się technikę zwaną programowaniem dynamicznym, która generalnie jest troszkę skomplikowana. W tym przypadku wystarczy jednak wykazać się tylko odrobiną sprytu: po co za każdym razem liczyć każdy wyraz od początku, skoro można zapisywać wyniki pośrednie? Nie będzie to zużywało wiele pamięci, gdyż musimy znać jedynie dwa poprzedzające wyrazy. Iteracyjny algorytm obliczania ciągu Fibonacciego może więc wyglądać tak:

```
unsigned Fib(unsigned n)
{
   unsigned uFib, uFib1, uFib2;

   // warunki brzegowe
   uFib = uFib1 = uFib2 = 1;

   // liczymy po kolei wyrazy aż do żądanego
   for (unsigned i = 3; i <= n; ++i)
   {
         // "przewijamy" dwa wyrazy poprzedzające
         uFib2 = uFib1;
         uFib1 = uFib;

         // obliczamy nowy wyraz
         uFib = uFib1 + uFib2;
   }

   // zwracamy wynik
   return uFib;
}
```

Różnica jest kolosalna. Z pobieżnego rzutu oka na pętlę w powyższej funkcji wynika bowiem, że jest ona rzędu zaledwie $\mathcal{O}(n)$! Wersja iteracyjna pozwala więc zejść ze złożoności wykładniczej do liniowej.

Rekurencja w technice „dziel i zwyciężaj”

Jeżeli nie miałeś wcześniej do czynienia z algorytmami rekurencyjnymi, to poprzednim paragrafem mogłem cię do nich „nieco” zniechęcić. Nie powinieneś jednak brać ich sobie za bardzo do serca. W rzeczy samej rekurencja nie jest wcale takim diabłem, jakim to trochę niechcący przedstawiłem ją wcześniej. Co ważniejsze, istnieje wiele problemów, dla których tylko algorytmy oparte na rekursji dają efektywne albo wręcz jedyne poprawne rozwiązanie.

Pokaźną grupę stanowią tutaj operacje na strukturach danych, które same w sobie mają naturę rekurencyjną. Przykładem mogą być grafy i drzewa, a praktycznym zastosowaniem - chociażby wyszukiwanie pliku o określonej nazwie w rozległym drzewie katalogów dyskowych.
Na rekurencji jest też oparta bardzo skuteczna i ogólna metoda projektowania algorytmów, znana jako **„dziel i zwyciężaj”** (ang. *divide & conquer*). Jej idea polega na podziale zadania na mniejsze i kontynuowanie tego procesu aż do momentu uzyskania problemu elementarnego, który potrafimy rozwiązać bezpośrednio. W sumie algorytm korzystający z tej metody składa się z trzech części:

- podzielenia problemu na mniejsze fragmenty (podproblemy)
- rozwiązania podproblemów poprzez dalszy, rekurencyjny podział - aż dojdziemy do przypadku elementarnego, „niepodzielnego”
- złączenia rozwiązań podproblemów w jedno rozwiązanie całego problemu

Opis ten może brzmi nieco zawile, lecz sama idea kryjąca się za nim jest w gruncie rzeczy bardzo prosta. Jak to czesto bywa, najlepiej zobaczyć ją na przykładzie. Ponownie pozwolę sobie na wykorzystanie w tym celu sortowania, jako że czynność ta jest dla komputerów zapewne tak samo naturalna, jak dla nas oddychanie czy jedzenie. Na początku rozdziału poznaliśmy prosty algorytm zajmujący się tym zadaniem i w chwilę potem określiliśmy jego złożoność jako $\Theta(n^2)$. Przy użyciu techniki „dziel i zwyciężaj" można ten czas obniżyć do $\Theta(n\log n)$, co jest czasem **optymalnym** (najmniejszym z możliwych) dla sortowania przez porównywanie. Oczywiście nas będzie również żywo interesowało to, jak tę złożoność wyznaczyć; tym także zajmiemy się za chwilę.

Przykład: sortowanie przez scalanie

Wpierw czas na obiecany przykład. Algorytmów sortowania skonstruowanych w oparciu o technikę „dziel i zwyciężaj" jest przynajmniej kilka, a my nie będziemy tutaj poznawać ich wszystkich, bo w końcu nie jest to książka o algorytmach jako takich, tylko rozdział poświęcony analizie ich złożoności. Zobaczymy więc jeden z nich, nazywany **sortowaniem przez scalanie** albo **złączanie** (ang. *mergesort*). Ma on tę zaletę, że jest całkiem prosty, a ponadto dobrze ilustruje sposoby na określanie złożoności algorytmów opartych o „dziel i zwyciężaj".

Zaatakowanie od razu kodem źródłowym byłoby pewnie nieco odstraszające, zatem najpierw pomówmy sobie, jak ten algorytm działa. Danymi wejściowymi jest naturalnie tablica liczb (lub dowolnych elementów, które można sortować, porównując ze sobą) o rozmiarze *n*. Dla takiej tablicy wykonywane są trzy poniższe kroki:

- *dziel*: tablica jest dzielona na pół, czyli na dwie podtablice o rozmiarze $n/2$
- *zwyciężaj*: każda podtablica jest sortowana poprzez rekurencyjne wywołanie algorytmu - z wyjątkiem tej o rozmiarze 1, którą siłą rzeczy jest od razu posortowana
- *połącz*: posortowane podtablice są następnie łączone w jedną posortowaną tablicę

Na pierwszy rzut oka działanie algorytmu może wydawać irracjonalne. Cóż nam z bowiem z samego rekurencyjnego dzielenia tablic na coraz mniejsze?… Oczywiście nic, jednak cała praca, dzięki której algorytm działa, jest wykonywana w kroku trzecim. Stąd właśnie wzięła się nazwa sortowania przez scalanie: łączenie posortowanych podtablic jest bowiem „punktem ciężkości" algorytmu - to jemu zawdzięczamy jego poprawność.

Z kolei drugi popularny algorytm sortowania - zwany tendencyjnie **sortowaniem szybkim** (ang. *quicksort*), choć bardziej odpowiednia nazwa to sortowanie przez podział - kładzie nacisk na krok pierwszy. Tam sposób podziału jest sprawą kluczową, natomiast operacja łączenia w ogóle nie jest potrzebna.

W sortowaniu przez scalanie to łączenie zajmuje więc najwięcej czasu i miejsca w kodzie źródłowym. Spójrzmy więc teraz, jak to wszystko wygląda:

```
// sortowanie przez scalanie
// parametry uMinIndex i uMaxIndex wyznaczają aktualnie sortowaną
// podtablicę
void MergeSort(const int* aTablica, unsigned uRozmiar,
               unsigned uMinIndex = 0,
               unsigned uMaxIndex = uRozmiar - 1)
{
   // najpierw sprawdzamy, czy podana tablica ma co najmniej
   // dwa elementy; jeżeli nie, jest to przypadek elementarny,
   // kończacy rekurencję
   if (uMinIndex >= uMaxIndex)   return;


   /* krok 1: dziel */
```

```
    // punkt podziału wyznaczamy w połowie tablicy
    unsigned uPunktPodzialu = (uMinIndex + uMaxIndex) / 2;


    /* krok 2: zwyciężaj */

    // wywołujemy rekurencyjnie procedurę dla obu połówek tablicy
    MergeSort (aTablica, uRozmiar, uMinIndex, uPunktPodzialu);
    MergeSort (aTablica, uRozmiar, uPunktPodzialu + 1, uMaxIndex);


    /* krok 3: połącz */

    // obliczamy rozmiary obu podtablic
    unsigned uRozmiarLewej = uPunktPodzialu - uMinIndex + 1;
    unsigned uRozmiarPrawej = uMaxIndex - uPunktPodzialu;

    // tworzymy pomocnicze tablice (o jeden element większe)
    int[] aLewa = new int [uRozmiarLewej + 1];
    int[] aPrawa = new int [uRozmiarPrawej + 1];

    // wypełniamy je zawartością odpowiednich połówek
    // (uwaga: w prawdziwej implementacji używamy nie pętli,
    //  lecz funkcji w rodzaju memcpy())
    for (unsigned i = 0; i < uRozmiarLewej; ++i)
          aLewa[i] = aTablica[uMinIndex + i];
    for (unsigned i = 0; i < uRozmiarPrawej; ++i)
          aPrawa[i] = aTablica[uPunktPodzialu + 1 + i];

    // dodajemy też wartowników na końcach: elementy większe od
    // dowolnego innego
    aLewa[uRozmiarLewej] = aPrawa[uRozmiarPrawej] = MAX_INT;

    // następnie bierzemy elementy raz z jednej, raz z drugiej
    // tablicy (zawsze mniejszy) i wstawiamy do oryginalnej
    unsigned i = 0, j = 0;
    for (unsigned k = uMinIndex; k <= uMaxIndex; ++k)
          // porównujemy elementy z obu tablic
          if (aLewa[i] <= aPrawa[j])
          {
                // lewa mniejsza; wstawiamy element, inkrem. i
                aTablica[k] = aLewa[i];
                ++i;
          }
          else
          {
                // prawa mniejsza; wstawiamy element, inkrem. j
                aTablica[k] = aPrawa[j];
                ++j
          }

    // na koniec pozbywamy się niepotrzebnych już tablic pomocniczych
    delete[] aLewa;
    delete[] aPrawa;
}
```

Tablicę dzielimy więc na pół i każdą połówkę sortujemy rekurencyjnie. Łączenie, większa część procedury, odbywa się natomiast w pewien sprytny sposób. Polega ono mianowicie na kolejnym sprawdzaniu elementów z obu podtablic i wybieraniu większego. Po opróżnieniu którejś podtablicy dodawana jest następnie cała zawartość drugiej. Dzięki

obecności wartowników na końcach, za wszystko odpowiada ostatnia pętla. Inna metoda polega na ręcznym sprawdzaniu, czy któraś z podtablic jest pusta; wtedy potrzebne są jeszcze dwie pętle (z których wykonuje się tylko jedna), które „opróżniają” drugą połówkę.

Analiza

Sortowanie przez scalanie wyglądać może na o wiele bardziej skomplikowane niż to przez wstawianie. Gra jest jednak warta świeczki - zyskiem jest wzrost efektywności całej procedury. Zajmijmy się zatem jej wyznaczeniem, czyli znalezieniem klasy algorytmu *mergesort*.

Procedura `MergeSort()` jest rekurencyjna, zatem analizę możemy podzielić na dwie części. Pierwsze zadanie to wyznaczenie złożoności pojedynczego wywołania funkcji. Drugi etap to ustalenie, jaki koszt wnosi tutaj rekurencja. Połączenie tych dwóch rezultatów da nam w wyniku klasę algorytmu.

Analiza algorytmu łączenia

W przypadku sortowania przez scalanie owo scalanie jest główną częścią każdego wywołania procedury. Najpierw zatem zajmiemy się właśnie tym fragmentem sortowania.

Sam algorytm łączenia (rozpoczynający się w kodzie źródłowym o komentarza „krok 3”) nie jest rekurecyjny. Jak można stwierdzić pobieżnym rzutem oka, jego istotę stanowią przede wszystkim pętle. Najważniejsza jest ostatnia, przebiegająca po całej sortowanej w danym momencie podtablicy (po wszystkich indeksach od `uMinIndex` do `uMaxIndex`). Wykonuje więc ona *n* cykli, gdzie *n* jest równe `uMaxIndex – uMinIndex + 1`. Jej złożoność jest liniowa - $\Theta(n)$.
Reszta algorytmu nie przekracza tej klasy. Wypełnienie wartościami dwóch pomocniczych tablic `aLewa` i `aPrawa` także wymaga liczby instrukcji proporcjonalnej do *n*. Jeśli zaś założymy, że operacje alokacji pamięci i jej zwalniania są wykonywane w czasie stałym (co jest rozsądne dla niemal wszystkich komputerów), to ze strony złożoności algorytmu łączenia podtablic nie spotkają nas już żadne niespodzianki.

Ostatecznie więc jest on rzędu $\Theta(n)$.

Złożoność teoretyczna

Wiemy teraz, jak efektywne jest zrealizowanie pojedynczego wywołania rekurencyjnego w sortowaniu *mergesort*. Nie wiemy jednak, ile takich wywołań rzeczywiście występuje i jak bardzo wielkość ta zależy od *n* - rozmiaru sortowanej tablicy. Ażeby to oszacować, musimy przyjrzeć się zastosowanej rekursji i w ten sposób wyznaczyć funkcję złożoności dla całego algorytmu.

Zobaczmy więc, jak *mergesort* wywołuje sam siebie. Z opisu podanego w poprzednim paragrafie powinieneś jeszcze pamiętać, że istotą jest tu podział tablicy na dwie połowy. I tak się faktycznie dzieje: wyznaczany jest po prostu graniczny „indeks połówkowy”, wedle którego dokonywany jest podział (przechowuje go zmienna `uPunktPodziału`).
A gdy dokonało się dzielenie, czas na zwycięstwo. Obie podtablice są więc sortowane poprzez ten sam algorytm *mergesort* - z tą różnicą, że jest on wywoływany dla każdej z nich **osobno**. Rekurencyjne wywołania procedury operują już zatem na tablicach o rozmiarze nie *n*, lecz mniej więcej[8] *n*/2.

---

[8] Żaden element nie może rzecz jasna zostać zgubiony. W rzeczywistości pierwsza rekurencja zajmuje się więc podtablicą o rozmiarze $\lfloor n/2 \rfloor$ (połowa liczby elementów zaokrąglona w doł), zaś druga - o rozmiarze $\lceil n/2 \rceil$ (połowa liczby element zaokrąglona w górę). Dla analizy algorytmu jest to jednak szczegół techniczny, bo skoro wszystkie elementy i tak są brane pod uwagę, możemy swobodnie założyć, że obie połówki mają po prostu *n*/2 elementów.

Wartość $T(n)$ złożoności praktycznej jest więc budowana przez wartości $T(n/2)$, reprezentujące rekurencję dla tablic połówkowych, oraz złożoność procesu łączenia - $\Theta(n)$. W sumie wynosi ona zatem:

$$T(n) = 2T(n/2) + \Theta(n)$$

Należy jeszcze uwzględnić przypadek elementarny - jest nim tablica składająca się tylko z jednego elementu. Naturalnie jest on posortowana, toteż do algorytmu należy jedynie stwierdzenie tego faktu. Jest to wykonywane w czasie stałym - $\Theta(1)$.

Finalna postać zależności $T(n)$ jest więc następująca:

$$T(n) = \begin{cases} \Theta(1) & \text{dla } n \le 1 \\ 2T(n/2) + \Theta(n) & \text{dla } n > 1 \end{cases}$$

Teraz pozostaje nam „tylko" jej rozwiązanie, czyli wyznaczenie $T(n)$ jako funkcji nierekurencyjnej. W następnym paragrafie zajmiemy się tym głównym punktem programu.

Jeżeli posługiwanie się notacjami asymptotycznymi w równaniach sprawia ci jeszcze kłopot, to możesz przyjąć, że pod $\Theta(1)$ kryje się c, zaś pod $\Theta(n)$ - dn, gdzie c i d są dowolnymi stałymi. Ja jednak będę stosował ten zapis jako wygodniejszy i podkreślający fakt, że zależy nam wyznaczeniu złożoności asymptotycznej bez wdawania się w zbędne szczegóły. Właściwie więc notację Θ należałoby tu traktować jako pewne uproszczenie!

Rozwiązanie rekurencji

Już pierwszy rzut oka na równanie

$$T(n) = 2T(n/2) + \Theta(n)$$

utwierdza nas w przekonaniu, że różni się ono trochę od tych, którymi zajmowaliśmy się dotąd. Pomijając występowanie więcej niż jednego składnika rekurencyjnego (z czym nauczyliśmy sobie jakoś radzić), zamieszanie wprowadza z pewnością $n/2$ jako jego argument.
Dzielenie rozmiaru danych na połowę lub dowolną inną liczbę części jest jednak (jak nawet sama nazwa wskazuje) nieodłącznym elementem techniki „dziel i zwyciężaj". Gdy więc poznamy sposób na rozwikłanie powyższej funkcji jest wielce prawdopodobne, że analogiczne metody dadzą się zastosować dla przynajmniej większości algorytmów stosując ww. technikę. W tym paragrafie pokażę kilka takich sposobów.

Na początek jednakowoż wypadałoby podjąć wyzwanie i oszacować powyższą funkcję $T(n)$ dla sortowania przez scalanie. Mimo pozornej trudności zadanie to może okazać się całkiem łatwe…

Jeszcze raz drzewko

Oczywiście metoda rozpisywania na pewno nie zda tu egzaminu, gdyż czynnik 2 przy $T(n/2)$ znakomicie uniemożliwia zredukowanie wszystkich składników poza $T(n)$. Poradziliśmy już sobie jednak w takiej sytuacji: pomocą okazało się zilustrowanie rekurencji za pomocą poglądowego drzewka.

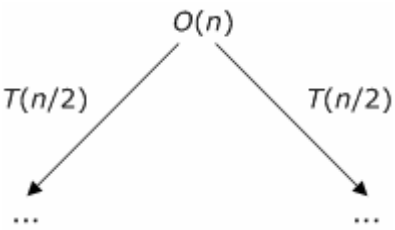
Spróbujmy więc wykorzystać tę metodę także i tutaj. Przypomnijmy, że drzewko jest skonstruowane wedle trzech reguł:

- każdy jego węzeł odpowiada nierekurencyjnej część równania - tej, którą znamy bezpośrednio, niezawierającej dalszych wywołań $T(...)$

- krawędź (gałąź) drzewa wychodząca z danego węzła reprezentuje wywołania rekurencyjne
- liście drzewa odpowiadają przypadkom elementarnym, kończącym rekurencję

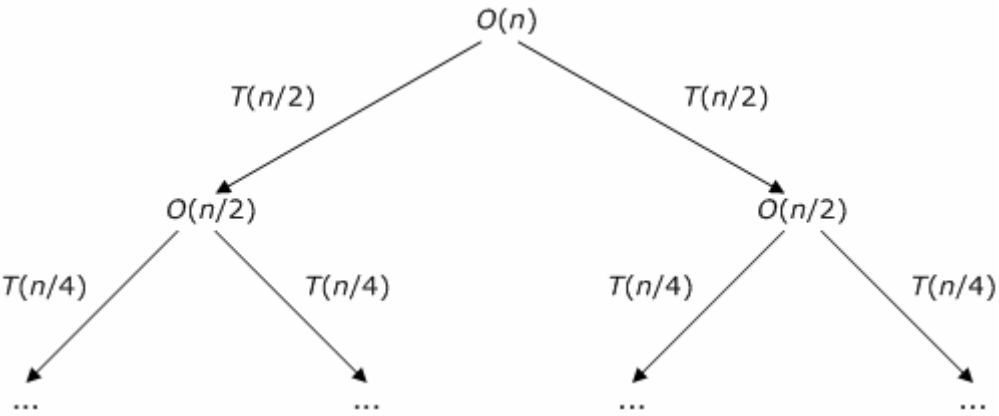
Jak to wygląda u nas?… Składnikiem nierekurencyjnym w $T(n)$ jest $\Theta(n)$ - przypomnijmy, że jest to synonim dowolnej funkcji liniowej. Pojawi się on więc w korzeniu i wewnętrznych węzłach drzewa. Z kolei krawędzie są modelem przywołań rekurencyjnych. Od korzenia odchodzą więc dwie gałęzie $T(n/2)$, na drugim poziomie - $T(n/4)$, potem $T(n/8)$, itd. Wreszcie, drzewo kończy się na przypadkach elementarnych, gdy n nie można już podzielić na dwa. W liściach znajdzie się więc $\Theta(1)$.

Opierając się na tych spostrzeżeniach możemy zasadzić drzewko:



Schemat 4. Drzewo rekursji dla złożoności teoretycznej sortowania przez scalanie - etap pierwszy

Pod wielokropkami kryją się oczywiście rekurencyjne poddrzewa. Gdy więc nasze drzewko urośnie nieco bardziej, wyglądać będzie mniej więcej tak:

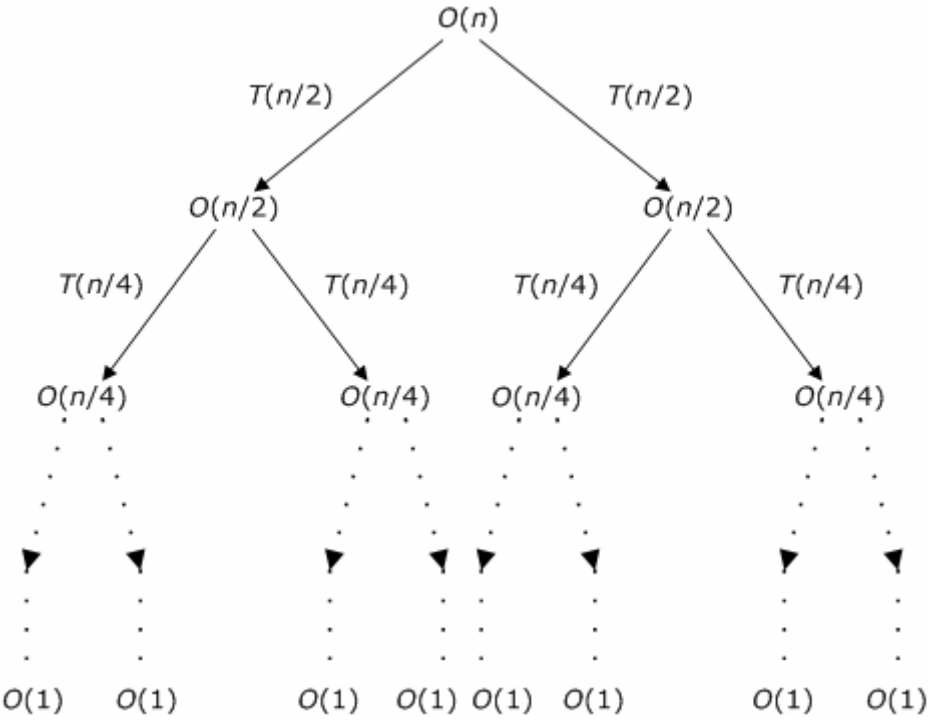


Schemat 5. Drzewo rekursji dla złożoności teoretycznej sortowania przez scalanie - etap drugi

Jak widzimy, kolejne wywołania są przeprowadzane dla coraz mniejszych wartości n - połówek, połówek połówek, połówek ćwierci, itd. W wyniku tego podziału dojdziemy w końcu do przypadku elementarnego $n = 1$. Wtedy też drzewko kończy się, a w liściu pozostaje jedynie składnik $\Theta(1)$.

Aby dzielenie przez 2 zredukowało w końcu n do samej jedynki, wartość n musi być oczywiście potęgą dwójki. Możemy bez przeszkód przyjąć takie założenie, gdyż nie wpływa ono na asymptotyczną złożoność algorytmu *mergesort*. Dla n nie będących potęgą dwójki drzewko nie będzie po prostu zrównoważone, czyli niektóre jego gałęzie będą kończyły się liśćmi wcześniej niż inne. Nie wpłynie to jednak na oszacowanie ilości węzłów w drzewie.

W pełni rozwnięte drzewo rekursji wygląda więc następująco:



Schemat 6. Gotowe drzewo rekursji dla złożoności teoretycznej sortowania przez scalanie

Może dziwić użycie wyrażeń w formie $\Theta(n/2)$ lub $\Theta(n/4)$, lecz ma to swoje uzasadnienie. Po zsumowaniu kosztów na każdym poziomie musimy bowiem otrzymać $\Theta(n)$, gdyż algorytm zajmuje się zawsze wszystkimi n elementami tablicy.

Na połączenie drzewka i notacji asymptotycznych trzeba więc nieco uważać. Generalnie jednak w rzeczywistej analizie algorytmów „dziel i zwyciężaj" w ogóle nie stosuje się drzew, lecz metody opisane w następnych akapitach. Dlatego też próba rozwiązania rekurencji za pomocą powyższego drzewka musi być traktowana trochę nieformalnie.

Powyższy fakt daje nam niespodziewanie cenną informację: dla każdego poziomu rekursji wykonywanych jest zawsze $\Theta(n)$ instrukcji. Celem oszacowania całkowitej złożoności musimy więc tylko znaleźć wysokość drzewka, a następnie pomnożyć tę wielkość przez $\Theta(n)$.
Ile poziomów rekursji występuje tutaj?… Odpowiedź jest prosta: tyle, ażeby z n „zejść" w końcu do 1 poprzez ciągłe dzielenie przez dwa (i zaokrąglanie w dół). A ponieważ każdy węzeł zajmuje się wartością dwa razy mniejszą niż węzeł nadrzędny (co odpowiada podziałowi tablicy na pół), więc na i-tym poziomie wartość ta wyniesie $n/2^{i-1}$. Jeśli zatem oznaczymy szukaną ilość poziomów jako p, to

$$\frac{n}{2^{p-1}} = 1$$

bo na ostatnim poziomie mamy już do czynienia jedynie z 1-elementową tablicą. Stąd można bez problemu wyznaczyć owe p:

$$p = \log_2 n + 1$$

Liczba poziomów drzewa zależy więc logarytmicznie od rozmiaru danych, czyli jest rzędu $\Theta(\log n)$. Wiemy również, że wykonanie każdego poziomu zajmuje czas $\Theta(n)$. Mnożąc obie wartości otrzymujemy całkowitą złożoność algorytmu:

$$\Theta(\log n)\cdot\Theta(n)=\Theta(n\log n)$$

Jest ona znacznie lepsza od $\Theta(n^2)$ sortowania przez wstawianie. Przykładowo posortowanie 1000 elementów zajmuje tamtą metodą około milion instrukcji, zaś sortowanie przez scalanie mniej więcej sto razy mniej. Tak efektywnośc ma jednak swoją cenę. Algorytm sortowania przez scalanie jest bardziej skomplikowany, podobnie jak „intuicyjna" analiza jego złożoności.

Metoda rekurencji uniwersalnej

Za pomocą umiejętnie użytego drzewka można rozwiązać prawie każdy problem analizy algorytmu typu „dziel i zwyciężaj". Dość często jest to jednak pracochłonne, wymaga też wyjątkowej uwagi i zwracania uwagi na takie niuanse, jak poprawne użycie notacji asymptotycznej.
Z drzewkami i rekurencją dawno temu walczyli już matematycy, a owocem ich pracy jest bardzo skuteczna metoda **rekurencji uniwersalnej**. Jej nazwa wskazuje, że można ją stosować do bardzo szerokiego zakresu funkcji i tak jest w istocie. Zaletą tej metody jest ponadto szybkość i względna łatwość stosowania. Nie potrzebujemy bowiem ani rozpisywania funkcji w szereg równań, ani rysowania drzewka. Wszystko, co jest potrzebne, to rodzaj „ściągi" pozwalającej na bezpośrednie określenie rozwiązania.

Zanim przedstawię tę metodę muszę jeszcze dokładnie określić zakres jej stosowalności. Otóż przy jej pomocy możemy w miarę prosty sposób określać klasę funkcji $T(n)$ występującej w równaniu postaci:

$$T(n)=aT(n/b)+f(n)$$

Współczynniki a i b są tu dowolnymi stałymi, zaś $f(n)$ - dowolną funkcją, określającą nierekurencyjną część równania (czyli nie zawierającą dalszych wywołań $T(...)$). Dla przykładu, nasze równanie określające złożoność sortowania przez scalanie ma współczynniki a i b równe 2, zaś $f(n)$ jest dowolną funkcję liniową.
Ponadto musimy oczywiście założyć, że rekurencja kiedyś się kończy, czyli dla wystarczającą małej wartości n określona jest terminalna stała.

W jaki sposób wygląda teraz zastosowanie metody rekurencji uniwersalnej? Składa się ono z dwóch kroków:

- najpierw należy obliczyć funkcję $g(n)=n^{\log_b a}$
- następnie należy **porównać** ją z funkcją $f(n)$ i na tej podstawie określić rozwiązanie

Pierwszy krok jest oczywiście bardzo prosty, szczególnie jeśli dysponujemy kalkulatorem czy innym urządzeniem liczącym. Dla, na przykład, sortowania przez scalanie wspomnianą funkcją będzie:

$$g(n)=n^{\log_2 2}=n^1=n$$

Podobnie jest dla dowolnych innych wartości a i b. To zdecydowanie prostszy krok tej metody.

Drugi krok polega na wyborze jednej z trzech możliwych rozwiązań w zależności od wyniku porównania funkcji $g(n)$ i $f(n)$. Gdy mówimy o porównywaniu funkcji, mamy oczywiście na myśli ich relacje wyrażone za pomocą poznanych notacji asymptotycznych: Ω, Θ i O. W tym przypadku również tak jest.
W metodzie rekurencji uniwersalnej mamy więc trzy przypadki, które przedstawia poniższa tabela:

relacja między g(n) i f(n)	*rozwiązanie*
$g(n)$ jest wielomianowo większa od $f(n)$	$T(n)=\Theta\left(g(n)\right)=\Theta\left(n^{\log_b a}\right)$
$g(n)$ jest asymptotycznie równa $f(n)$	$T(n)=\Theta\left(g(n)\cdot\log n\right)=\Theta\left(n^{\log_b a}\cdot\log n\right)$
$g(n)$ jest wielomianowo mniejsza od $f(n)$	$T(n)=\Theta\left(f(n)\right)$

Tabela 4. Trzy możliwe przypadki w metodzie rekurencji uniwersalnej

Należy po prostu stwierdzić, który z nich zachodzi, a potem bezpośrednio odczytać rozwiązanie… Cóż, łatwiej powiedzieć, ale pewnie trudniej zrobić. Wyjaśnienia wymaga na pewno określenie, że funkcja jest „**wielomianowo**" większa lub mniejsza od innej. Stwierdzenie to oznacza mianowicie, że obie funkcje muszą różnić się od siebie przynajmniej o **czynnik wielomianowy** - tzn. o n^k, gdzie k jest dowolną liczbą dodatnią. Weźmy np. $f(n) = \Theta(n\log n)$ i $g(n) = \Theta(n^2)$. Tutaj wiadomo rzecz jasna, że $g(n)$ jest większa od $f(n)$, jednak nie jest ona wielomianowo większa. Różnica między obiema funkcjami sprowadza się bowiem do czynnika logarytmicznego - log$n$ - który jest mniejszy niż wielomianowy. W takiej sytuacji jak powyższa **nie moglibyśmy** niestety zastosować metody rekurencji uniwersalnej; wariant ten niejako „wpada w lukę" pomiędzy przypadkami 1 i 2.

Na szczęście w większości równań obie funkcje spełniają któryś z trzech warunków. Dla naszego sortowania zachodzi na przykład przypadek drugi, gdyż zarówno $f(n)$, jak i $g(n)$ są sobie asymptotycznie równe: obie to funkcje liniowe. Wynika stąd natychmiast, że $T(n)$ jest rzędu $\Theta(g(n)\log n)$, czyli $\Theta(n\log n)$. Uzyskaliśmy więc taki sam wynik jak przy zastosowaniu drzewka, jednak metoda rekurencji uniwersalnej jest zwykle o wiele szybsza i wygodniejsza.

Ciekawostka: twierdzenie o rekurencji uniwersalnej

Cała ta metoda opiera się na matematycznym **twierdzeniu o rekurencji uniwersalnej**. Jego treść jest zgodna z informacjami z poprzedniego akapitu, aczkolwiek formalny język matematyki czyni ją nieco precyzyjniejszą. Naturalnie nie ma najmniejszej potrzeby, abyś znał je na pamięć; wystarczy tylko byś wiedział, gdzie możesz je znaleźć i jak je zastosować.

Oto więc rzeczone twierdzenie[9]:

Niech $a \geq 1$ i $b > 1$ będą stałymi, $f(n)$ dowolną funkcją, zaś $T(n)$ zdefiniowane dla nieujemnych liczb całkowitych poprzez rekurencję:

$$T(n)=aT(n/b)+f(n)$$

Wówczas $T(n)$ możemy asymptotycznie oszacować w następujący sposób:

1. Jeśli $f(n)=O\left(n^{\log_b a-\varepsilon}\right)$ dla pewnej stałej $\varepsilon > 0$, to $T(n)=\Theta\left(n^{\log_b a}\right)$
2. Jeśli $f(n)=\Theta\left(n^{\log_b a}\right)$, to $T(n)=\Theta\left(n^{\log_b a}\log n\right)$
3. Jeśli $f(n)=\Omega\left(n^{\log_b a+\varepsilon}\right)$ dla pewnej stałej $\varepsilon > 0$, to $T(n)=\Theta\left(f(n)\right)$

[9] Cytowane za *Wprowadzeniem do algorytmów* Thomasa H. Cormena. Pominąłem jedynie tzw. warunek regularności w trzecim przypadku, gdyż jest on spełniony dla wszystkich rozsądnych funkcji pojawiających się w analizie algorytmów.

Najprawdopodobniej stwierdzisz, że wygląda ono dość upiornie. Generalnie jednak jest to dokładnie to samo zestawienie trzech możliwych przypadków, podanych w tabeli z poprzedniego akapitu. Można jeszcze zwrócić uwagę, jak została zapisana relacja wielomianowa: poprzez wprowadzenie stałej $\varepsilon > 0$ do wykładnika n w funkcji $g(n) = n^{\log_b a}$.

Podsumowanie

Analiza efektywności algorytmów może nie wydwać się wdzięcznym ani prostym zadaniem. Przynajmniej podstawowa jej znajomość jest jednak konieczna, aby móc pisać programy, które nie będą działały ślamazarnie nawet na najszybszych komputerach. Ponadto całkiem często okazuje się, że niemożność określenia złożoności własnego algorytmu staje się silną przesłanką za tym, iż jest on błędny lub w najlepszym wypadku mało wydajny.

W tym rozdziale mogłeś więc poznać garść wiedzy na temat tego ważnego zagadnienia, jakim jest wyznaczanie szybkości algorytmów. Najpierw więc zastanowiliśmy się, w jaki sposób można rozsądnie wyrażać efektywność danego algorytmu i jakich miar można użyć do porównywania wydajności procedur. Zwróciłem wówczas uwagę, że przyglądanie się wszelkim sprawom „technicznym” prowadzi zwykle donikąd.
Potem poznaliśmy najbardziej rozpowszechnioną metodę wyrażania sprawności algorytmów, czyli złożoność teoretyczną. Ku radości nielicznych i narzekaniu większości zagłebiliśmy się też w matematyczną stronę tego zagadnienia, a mianowicie notacje asymptotyczne.
Było to jednak konieczne, abyśmy mogli przejśc do zasadniczej treści rozdziału. W drugiej jego części przedstawiłem więc typowe techniki służące znajdowaniu złożoności różnych algorytmów. Sporo uwagi poświęciliśmy pętlom, które występują w prawie każdej procedurze. Ponadto zajęliśmy się także programami rekurencyjnymi, które wprawdzie nie występują już często, jednak są o wiele oporniejsze w analizie. Tutaj także wymagana była porcja matematyki „wyższej”, czego ukoronowaniem była metoda rekurencji uniwersalnej.

Zapewne zdajesz sobie sprawę, że podane przeze mnie wskazówki absolutnie nie wyczerpują tematu badania efektywności algorytmów. Istnieje mnóstwo źródeł opisujących tę tematykę, z których największe znaczenie mają pozycje książkowe poświęcone algorytmice.

Pytania i zadania

Dla utrwalenia zdobytych wiadomości i umiejętności zalecane jest wykonanie poniższych ćwiczeń i odpowiedz na pytania. Powodzenia!

Pytania

1. Dlaczego podanie czasu wykonania procedury niewiele mówi o jej faktycznej efektywności?
2. Czym jest rozmiar danych algorytmu? Podaj kilka typowych przykładów.
3. Jakie instrukcje zwykle uważamy za elementarne?
4. Co to jest złożoność praktyczna algorytmu?
5. Jakie trzy przypadki działania można rozważać dla każdego algorytmu? Który z nich najbardziej się liczy i dlaczego?
6. Czym jest złożoność teoretyczna (klasa) algorytmu?
7. Dlaczego przy podawaniu klasy algorytmu stosujemy notacje asymptotyczne?

8. Jakie złożoności mają algorytmy, o których możemy powiedzieć, że są „efektywne”?
9. Jakie dwie szczególne cechy mają problemy NP-zupełne?
10. Jakie dwie ogólne zasady mają zastosowanie przy wyznaczaniu klasy algorytmów?
11. Jaką złożoność ma *k*-krotnie zagnieżdżona pętla przebiegająca po wszystkich wartościach *n* cyklami o stałym czasie?
12. Czym jest rekurencja?
13. Jakie dwie metody możemy spróbować zastosować do oszacowania prostych funkcji rekurencyjnych?
14. Na czym polega technika projektowania algorytmów znana jako „dziel i zwyciężaj”?
15. Jakie dwa sposoby można zastosować do szacowania efektywności algorytmów wykorzystujących tę technikę?
16. Co to znaczy, że jedna funkcja jest wielomianowo większa od drugiej?

Ćwiczenia

1. Podaj jakie czynniki techniczne, oprócz tych wymienionych na początku pierwszego podrozdziału, mogłyby jeszcze wpływać na szybkość wykonywania się algorytmu w rzeczywistym programie.
2. Udowodnij, że $O\left(f\left(n\right)\right) \cap \Omega\left(f\left(n\right)\right) = \Theta\left(f\left(n\right)\right)$.
3. Znajdź najmniejszą wartość *n*, dla której algorytm o złożoności praktycznej n^{16} byłby mimo wszystko szybszy od tego o złożoności 2^n.
4. Algorytm wyszukiwania binarnego służy do wyszukiwania podanej wartości w posortowanej tablicy. Działa on w ten sposób, że dla podanej tablicy porównuje szukany element z jej elementem środkowym i zależnie od wyniku wywołuje rekurencyjnie sam siebie dla lewej lub prawej połowy. Określ klasę tego algorytmu (dowolną metodą); być może koniecznie będzie zapisanie go w postaci (pseudo)kodu.