# Czy wszystko można policzyć na komputerze

**Rodzaj zajęć:** Wszechnica Popołudniowa
**Tytuł:** Czy wszystko można policzyć na komputerze
**Autor:** prof. dr hab. Maciej M Sysło

**Redaktor merytoryczny:** prof. dr hab. Maciej M Sysło

Zeszyt dydaktyczny opracowany w ramach projektu edukacyjnego **Informatyka+** — ponadregionalny program rozwijania kompetencji uczniów szkół ponadgimnazjalnych w zakresie technologii informacyjno-komunikacyjnych (ICT).
**www.informatykaplus.edu.pl**
**kontakt@informatykaplus.edu.pl**

**Wydawca:** Warszawska Wyższa Szkoła Informatyki
ul. Lewartowskiego 17, 00-169 Warszawa
**www.wwsi.edu.pl**
**rektorat@wwsi.edu.pl**

Projekt graficzny: FRYCZ I WICHA

Warszawa 2010


KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

# Czy wszystko można policzyć na komputerze

**Maciej M. Sysło**
Uniwersytet Wrocławski, UMK w Toruniu
syslo@ii.uni.wroc.pl, syslo@mat.uni.torun.pl

**Streszczenie**

Przedmiotem wykładu jest łagodne wprowadzenie do złożoności problemów i algorytmów. Przewodnim pytaniem jest, jak dobrze sprawują się algorytmy i komputery i czy komputery już mogą wszystko obliczyć. Z jednej strony, dla niektórych problemów (jak znajdowanie najmniejszego elementu) znane są algorytmy, które nie mają konkurencji, gdyż są bezwzględnie najlepsze, a z drugiej – istnieją problemy, o których przypuszcza się, że komputery nigdy nie będą w stanie ich rozwiązywać dostatecznie szybko. Przedstawione zostaną problemy, dla których są znane algorytmy optymalne (tj. takie, których nie można już przyspieszyć), oraz takie problemy, których nie potrafimy rozwiązywać szybko, nawet z użyciem najszybszych komputerów. Problemy z tej drugiej grupy znajdują zastosowanie na przykład w kryptografii. Rozważania będą ilustrowane praktycznymi zastosowaniami omawianych problemów i ich metod obliczeniowych.

Rozważania są prowadzone na elementarnym poziomie i do ich wysłuchania wystarczy znajomość informatyki wyniesiona z gimnazjum. Te zajęcia są adresowane do wszystkich uczniów w szkołach ponadgimnazjalnych, zgodnie bowiem z nową podstawą programową, kształceniem umiejętności algorytmicznego rozwiązywania problemów mają być objęci wszyscy uczniowie.

**Spis treści**

## 1 WPROWADZENIE

Można odnieść wrażenie, że komputery są obecnie w stanie wykonać wszelkie obliczenia i dostarczyć oczekiwany wynik w krótkim czasie. Budowane są jednak coraz szybsze komputery, co może świadczyć o tym, że moc istniejących komputerów nie jest jednak wystarczająca do wykonania wszystkich obliczeń, jakie nas interesują, potrzebne są więc jeszcze szybsze maszyny.

Zgodnie z nieformalnym prawem Moore'a, szybkość działania procesorów stale rośnie i podwaja się co 24 miesiące. Jednak istnieje wiele trudno rozwiązywalnych problemów, których obecnie nie jesteśmy w stanie rozwiązać za pomocą żadnego komputera i zwiększanie szybkości komputerów niewiele zmienia tę sytuację. Kluczowe staje się więc opracowywanie coraz szybszych algorytmów. Jak to ujął Ralf Gomory, szef ośrodka badawczego IBM:

*Najlepszym sposobem przyspieszania komputerów*
*jest obarczanie ich mniejszą liczbą działań.*

To „obarczanie komputerów" coraz mniejszą liczbą działań należy odczytać, jako stosowanie w obliczeniach komputerowych coraz szybszych algorytmów.

## 2 SUPERKOMPUTERY I ALGORYTMY

### Superkomputery

Zamieścimy tutaj podstawowe informacje dotyczące najszybszych komputerów. Komputery, które w danej chwili lub w jakimś okresie mają największą moc obliczeniową przyjęło się nazywać **superkomputerami**. Prowadzony jest ranking najszybszych komputerów świata (patrz strona http://www.top500.org/), a faktycznie odbywa się wyścig wśród producentów superkomputerów i dwa razy w roku są publikowane listy rankingowe. Szybkość działania komputerów jest oceniana na specjalnych zestawach problemów, pochodzących z algebry liniowej – są to na ogół układy równań liniowych, złożone z setek tysięcy a nawet milionów równań i niewiadomych. Szybkość komputerów podaje się w jednostkach zwanych **FLOPS** (lub **flops** lub **flop/s**) – jest to akronim od ***FL****oating point* ***O****perations* ***P****er* ***S****econd*, czyli zmiennopozycyjnych operacji na sekundę. Dla uproszczenia można przyjąć, że FLOPS to są działania arytmetyczne (dodawanie, odejmowanie, mnożenie i dzielenie) wykonywane na liczbach dziesiętnych z kropką. W użyciu są jednostki: YFlops (yotta flops) – $10^{24}$ operacji na sekundeę, ZFlops (zetta flops) – $10^{21}$, EFlops (exa flops) – $10^{18}$, PFlops (peta flops) – $10^{15}$, TFlops (tera flops) – $10^{12}$, GFlops (giga flops) –$10^{9}$, MFlops (mega flops) – $10^{6}$, KFlops (kilo flops) – $10^{3}$.

W rankingu z listopada 2009 roku, najszybszym komputerem świata był **Cray XT Jaguar** firmy Cray, którego moc jest 1.75 PFlops, czyli wykonuje on ponad $10^{15}$ operacji na sekundę. Komputer ten pracuje w Department of Energy's Oak Ridge National Laboratory w Stanach Zjednoczonych. Komputer Jaguar jest zbudowany z 224162 procesorów Opteron firmy AMD. Drugie miejsce zajmuje komputer Roadrunner firmy IBM o mocy 1.04 PFlops. Dwa dalsze miejsca zajmują również komputery Cray i IBM a w pierwszej dziesiątce są jeszcze dwa inne komputery IBM.

Na początku 2010 roku najszybszym procesorem PC był Intel Core i7 980 XE o mocy 107.6 GFlops. Większą moc mają procesory graficzne, np. Tesla C1060 GPU (nVidia) ma moc 933 GFlops (w pojedynczej dokładności), a HemlockXT 5970 (AMD) osiąga moc 4640 GFlops (w podwójnej dokładności).

Bardzo dużą moc osiągają obliczenia wykonywane na rozproszonych komputerach osobistych połączonych ze sobą za pomocą Internetu. Na przykład, ma początku 2010 roku, Folding@Home osiągnął moc 3.8 PFlops, a komputery osobiste pracujące w projekcie sieciowym GIMPS nad znalezieniem coraz większej liczby pierwszej osiągnęły moc 44 TFlops.

Przy obecnym tempie wzrostu możliwości superkomputerów przewiduje się, że moc 1 EFlops ($10^{18}$ operacji na sekundę) zostanie osiągnięta w 2019 roku, natomiast firma Cray ogłosiła, że będzie w stanie zbudować komputer o tej mocy w 2010 roku. Naukowcy oceniają, że dla pełnego modelowania zmian pogody na Ziemi w ciągu dwóch tygodni jest potrzebny komputer o mocy 1 ZFlops ($10^{21}$ operacji na sekundę). Przewiduje się, że taki komputer powstanie przed 2030 rokiem.

W dalszych rozważaniach będziemy przyjmować, że dysponujemy najszybszym komputerem, który ma moc 1 PFlops, czyli wykonuje $10^{15}$ = 1 000 000 000 000 000 operacji na sekundę.

**Superkomputery a algorytmy**

Ciekawi nas teraz, jak duże problemy możemy rozwiązywać za pomocą komputera o mocy 1 PFlops. W tabeli 1 zamieściliśmy czasy obliczeń wykonanych na tym superkomputerze posługując się algorytmami o różnej złożoności obliczeniowej (pracochłonności) i chcąc rozwiązywać problemy o różnych rozmiarach.

*Tabela 1.*
Czasy obliczeń za pomocą algorytmów o podanej złożoności, wykonywanych na superkomputerze o mocy 1 PFlps, czyli wykonującym $10^{15}$ operacji na sekundę (« 1 sek. oznacza w tabeli, że czas obliczeń stanowił niewielki ułamek sekundy); parametr *n* określa rozmiar danych

| Złożoność algorytmu | $n = 100$ | $n = 500$ | $n = 1000$ | $n = 10000$ |
|---|---|---|---|---|
| log *n* | « 1 sek. | « 1 sek. | « 1 sek. | « 1 sek. |
| n | « 1 sek. | « 1 sek. | « 1 sek. | « 1 sek. |
| *n* log *n* | « 1 sek. | « 1 sek. | « 1 sek. | « 1 sek. |
| $n^2$ | « 1 sek. | « 1 sek. | 0,000000001 sek. | 0,0000001 sek. |
| $n^5$ | 0,00001 sek. | 0,03125 sek. | 1 sek. | 1,15 dni |
| $2^n$ | $4*10^7$ lat | $1{,}038*10^{128}$ lat | $3{,}3977*10^{278}$ lat | |
| *n*! | $2{,}959*10^{135}$ lat | | | |

Wyraźnie widać, że dwa ostatnie w tabeli 1 algorytmy są całkowicie niepraktyczne nawet dla niewielkiej liczby danych ($n = 100$). Istnieją jednak problemy, dla których wszystkich możliwych rozwiązań może być $2^n$ lub *n*!, zatem nawet superkomputery nie są w stanie przejrzeć wszystkich możliwych rozwiązań w poszukiwaniu najlepszego, a zdecydowanie szybszymi metodami dla tych problemów nie dysponujemy.

Można łatwo przeliczyć, że gdyby nasz superkomputer był szybszy, na przykład miał moc 1 ZFlops ($10^{21}$ operacji na sekundę), co ma nastąpić dopiero ok. roku 2030, to dwa ostatnie wiersze w tabeli 1 uległyby tylko niewielkim zmianom, nie na tyle jednak, by móc stosować dwa ostatnie algorytmy w celach praktycznych.

W następnym rozdziale przedstawimy kilka przykładowych problemów, dla których moc obliczeniowa superkomputerów może nie być wystarczająca, by rozwiązywać je dla praktycznych rozmiarów danych. Ma to złe i dobre strony. Złe – bo nie jesteśmy w stanie rozwiązywać wielu ważnych i istotnych dla człowieka problemów, takich np. jak prognozowanie pogody, przewidywanie trzęsień Ziemi i wybuchów wulkanów, czy też planowanie optymalnych podróży lub komunikacji. Dobre zaś – bo możemy wykorzystywać trudne obliczeniowo problemy w metodach szyfrowania, dzięki czemu nasz przeciwnik nie jest w stanie, nawet posługując się najszybszymi komputerami, deszyfrować naszych wiadomości, chociaż my jesteśmy w stanie szybko je kodować i wysyłać.

## 3 PRZYKŁADY TRUDNYCH PROBLEMÓW

Podamy tutaj kilka dość prostych problemów, których rozwiązywanie może nastręczać pewne trudności nawet z użyciem najszybszych komputerów.

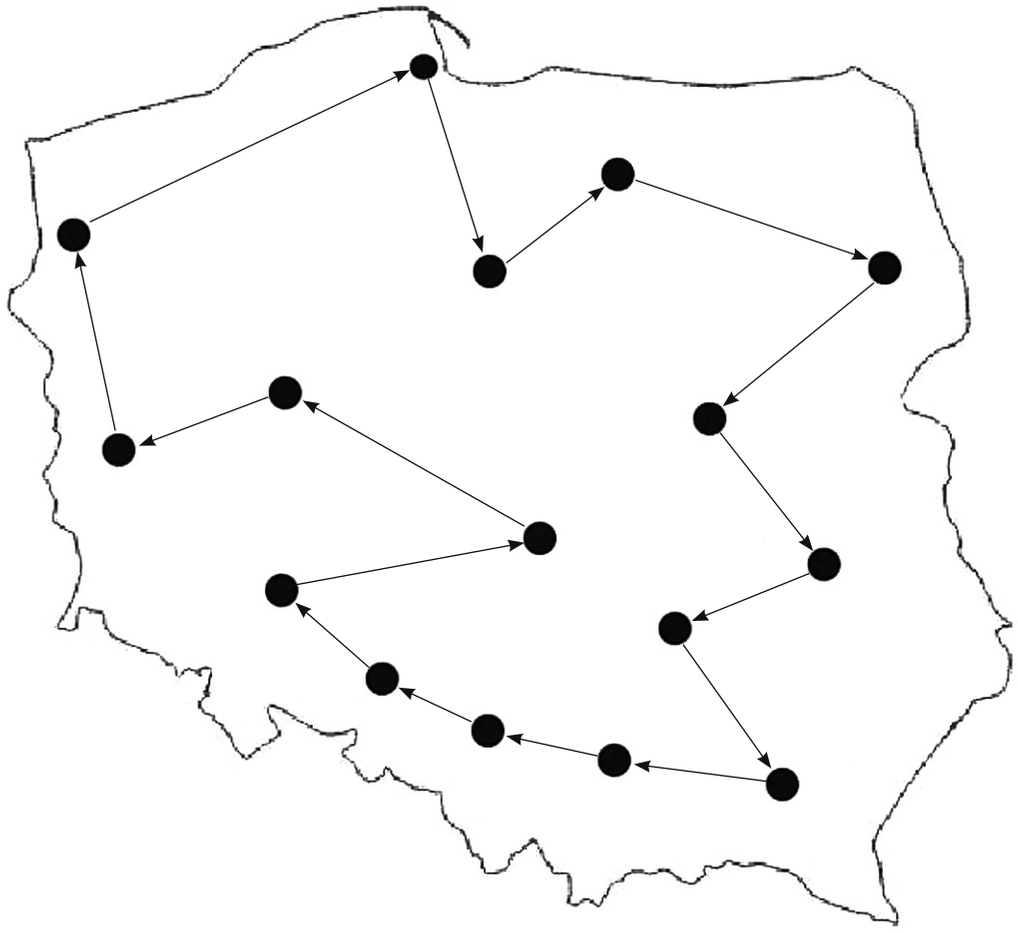
W tym rozdziale, jak i w dalszych, problemy definiujemy w postaci **specyfikacji**, która zawiera ścisły opis *Danych* i oczekiwanych *Wyników*. W specyfikacji są też zawarte zależności między danymi i wynikami. Specyfikacja problemu jest również specyfikacją algorytmu, który ten problem rozwiązuje, co ma na celu dokładne określenie przeznaczenia algorytmu, stanowi zatem powiązanie algorytmu z rozwiązywanym problemem.

### 3.1 NAJKRÓTSZA TRASA ZAMKNIĘTA

Jednym z najbardziej znanych problemów dotyczących wyznaczania tras przejazdu, jest **problem komiwojażera**, oznaczany zwykle jako **TSP**, od oryginalnej nazwy **Travelling Salesman Problem**. W tym problemie mamy dany zbiór miejscowości oraz odległości między nimi. Należy znaleźć drogę zamkniętą, przechodzącą przez każdą miejscowość dokładnie jeden raz, która ma najkrótszą długość.

Przykładem zastosowania problemu TSP może być zadanie wyznaczenia najkrótszej trasy objazdu prezydenta kraju po wszystkich stolicach województw (stanów – w Stanach zjednoczonych, landów – w Niemczech itp.). Na tej trasie, prezydent wyjeżdża ze stolicy kraju, ma odwiedzić stolicę każdego województwa dokładnie jeden raz i wrócić do stolicy kraju. Zapiszmy specyfikację tego problemu.

**Problem komiwojażera (TSP)**
*Dane:* *n* miast (punktów) i odległości między każdą parą miast.
*Wyniki:* Trasa zamknięta, przechodząca przez każde miasto dokładnie jeden raz, której długość jest możliwie najmniejsza.



*Rysunek 1.*
Przykładowa trasa przejazdu prezydenta po stolicach województw

Na rysunku 1 przedstawiliśmy jedną z możliwych tras, ale nie jesteśmy pewni, czy jest ona najkrótsza. Obsługa biura prezydenta może jednak chcieć znaleźć najkrótszą trasę. W tym celu postanowiono generować wszystkie możliwe trasy – zastanówmy się, ile ich jest. To łatwo policzyć. Z Warszawy można się udać do jednego z 15 miast wojewódzkich. Będąc w pierwszym wybranym mieście, do wyboru mamy jedno z 14 miast. Po wybraniu drugiego miasta na trasie, kolejne miasto można wybrać spośród 13 miast i tak dalej. Gdy osiągamy ostatnie miast, to czeka nas tylko powrót do Warszawy. A zatem wszystkich możliwych wyborów jest: 15*14*13*...*2*1. Oznaczmy tę liczbę następująco:

$$15! = 15*14*13*...*2*1$$

a ogólnie

$$n! = n*(n-1)*(n-2)*...*2*1$$

Oznaczenie *n*! czytamy **„n silnia”**, a zatem *n*! jest iloczynem kolejnych liczb całkowitych, od jeden do *n*. Wartości tej funkcji dla kolejnych *n* rosną bardzo szybko, patrz tabela 2.

*Tabela 2.*
Wartości funkcji *n*!

| *n* | *n*! |
|---|---|
| 10 | 3628800 |
| 15 | $1.30767*10^{12}$ |
| 20 | $2.4329*10^{18}$ |
| 25 | $1.55112*10^{25}$ |
| 30 | $2.65253*10^{32}$ |
| 40 | $8.15915*10^{47}$ |
| 48 | $1.24139*10^{61}$ |
| 100 | $9.3326*10^{157}$ |

Z wartości umieszczonych w tabeli 2 wynika, że posługując się superkomputerem w realizacji naszkicowanej metody, służącej do znalezienia najkrótszej trasy dla prezydenta Polski, otrzymanie takiej trasy zabrałoby mniej niż sekundę. Jednak w olbrzymim kłopocie znajdzie się prezydent Stanów Zjednoczonych chcąc taką samą metodą znaleźć najkrótszą trasę objazdu po stolicach wszystkich kontynentalnych stanów (jest ich 49, z wyjątkiem Hawajów). Niewiele zmieni jego sytuację przyspieszenie superkomputerów.

Znane są metody rozwiązywania problemu komiwojażera szybsze niż naszkicowana powyżej, jednak problem TSP pozostaje bardzo trudny. W takich przypadkach często są stosowane metody, które służą do szybkiego znajdowania rozwiązań przybliżonych, nie koniecznie najkrótszych. Jedna z takich metod, zwana **metodą najbliższego sąsiada**, polega na przykład na przejeżdżaniu w każdym kroku do miasta, które znajduje się najbliżej miasta, w którym się znajdujemy. W rozwiązaniu naszego problemu tą metodą, pierwszym odwiedzonym miastem powinna być Łódź, później Kielce, Lublin, Rzeszów, .., a nie jak na rys. 1 mamy Lubli, Rzeszów, Kraków ..., a Łódź gdzieś w trakcie podróży. Trasa otrzymana metodą najbliższego sąsiada jest krótsza niż trasa naszkicowana na rys. 1. W ogólnym przypadku ta metoda nie gwarantuje, że zawsze znajduje najkrótszą trasę.

### 3.2 ROZKŁAD LICZBY NA CZYNNIKI PIERWSZE

Liczby pierwsze stanowią w pewnym sensie „pierwiastki" wszystkich liczb, każdą bowiem liczbę całkowitą można jednoznacznie przedstawić, z dokładnością do kolejności, w postaci iloczynu liczb pierwszych. Na przykład, 4 = 2*2; 10 = 2*5; 20 = 2*2*5; 23 = 23; dla liczb pierwszych te iloczyny składają się z jednej liczby.

Matematycy interesowali się liczbami pierwszymi od dawna. Pierwsze spisane rozważania i twierdzenia dotyczące tych liczb znajdujemy w działach Euklidesa. Obecnie liczby pierwsze znajdują ważne zastosowania w kryptografii, m.in. w algorytmach szyfrujących. Do najważniejszych pytań, problemów i wyzwań, związanych z liczbami pierwszymi, należą następujące zagadnienia, które krótko komentujemy:

1. Dana jest dodatnia liczba całkowita *n* – czy *n* jest liczbą pierwszą (złożoną)?
   Ten problem ma bardzo duże znaczenie zarówno praktyczne (w kryptografii), jak i teoretyczne. Dopiero w 2002 roku został podany algorytm, który jednak jest interesujący głównie z teoretycznego punktu widzenia. Jego złożoność, czyli liczba wykonywanych operacji, zależy wielomianowo od rozmiaru liczby *n*, czyli od liczby bitów potrzebnych do zapisania liczby *n* w komputerze (ta liczba jest równa logarytmowi przy podstawie 2 z *n*). „Słabą" stroną większości metod, które udzielają odpowiedzi na pytanie: „czy *n* jest liczba pierwszą czy złożoną" jest udzielanie jedynie odpowiedzi „Tak" lub „Nie". Na ogół najszybsze metody dające odpowiedź na pytanie, czy *n* jest liczbą złożoną, w przypadku odpowiedzi „Tak" nie podają dzielników liczby *n* – dzielniki jest znacznie trudniej znaleźć niż przekonać się, że liczba jest złożona.
2. Dana jest dodatnia liczba całkowita *n* – rozłóż *n* na czynniki pierwsze.
   Ten problem ma olbrzymie znaczenie w kryptografii. Odpowiedź „Nie" udzielona na pytanie nr 1 nie pomaga w rozwiązaniu tego problemu. Z drugiej strony, możemy próbować znaleźć dzielniki liczby *n* dzieląc ją przez kolejne liczby, ale ta metoda jest mało praktyczna, gdyż w kryptografii występują liczby *n*, które mają kilkaset cyfr. Piszemy o tym dokładniej w punkcie 5.1.
3. Dana jest dodatnia liczba całkowita *m* – znajdź wszystkie liczby pierwsze mniejsze lub równe *m*.

To zadanie zyskało swoją popularność dzięki algorytmowi pochodzącemu ze Starożytności, który jest znany jako **sito Eratosthenesa**. Generowanie kolejnych liczb pierwszych tą metodą ma jednak niewielkie znaczenie praktyczne i jest uznawane raczej za ciekawostkę.

4. Znajdź największą liczbę pierwszą, a faktycznie, znajdź liczbę pierwszą większą od największej znanej liczby pierwszej. (Zgodnie z twierdzeniem Euklidesa, liczb pierwszych jest nieskończenie wiele, a zatem nie istnieje największa liczba pierwsza.)

   To wyzwanie cieszy się olbrzymią popularnością. Uruchomiony jest specjalny serwis internetowy pod adresem http://www.mersenne.org/, który jest wspólnym przedsięwzięciem sieciowym wielu poszukiwaczy. Obecnie największą liczbą pierwszą jest liczba $2^{43112609} - 1$, będąca liczbą Mersennea. Liczba ta ma 12 978 189 cyfr. Zapisanie jej w edytorze teksu (75 cyfr w wierszu, 50 wierszy na stronie) zajęłoby 3461 stron.

   W punkcie 5.1 wracamy do problemów nr 1 oraz 2 i przytaczamy znany algorytm badania, czy $n$ jest liczbą pierwszą. Jeśli $n$ jest liczbą złożoną, to możliwe jest w tym algorytmie generowanie kolejnych dzielników liczby $n$.

### 3.3 PODNOSZENIE DO POTĘGI

Podnoszenie do potęgi jest bardzo prostym, szkolnym zadaniem. Na przykład, aby obliczyć $3^4$, wykonujemy trzy mnożenia 4*4*4*4. A zatem w ogólności, aby obliczyć szkolną metodą wartość $x^n$, należy wykonać $n - 1$ mnożeń, o jedno mniej niż wynosi wykładnik potęgi. Czy ten algorytm jest na tyle szybki, by obliczyć na przykład wartość:

$$x^{12345678912345678912345678912345}$$

która może pojawić przy szyfrowaniu informacji przesyłanych w Internecie? Odpowiemy na to pytanie w punkcie 5.2.

### 3.4 PORZĄDKOWANIE

Problem **porządkowania**, często nazywany **sortowaniem**, jest jednym z najważniejszych problemów w informatyce i w wielu innych dziedzinach. Jego znaczenie jest ściśle związane z zarządzaniem danymi (informacjami), w szczególności z wykonywaniem takich operacji na danych, jak wyszukiwanie konkretnych danych lub umieszczanie danych w zbiorze.

Zauważmy, że mając $n$ elementów, które chcemy uporządkować, istnieje $n!$ możliwych uporządkowań, wśród których chcemy znaleźć właściwe uporządkowanie. A zatem, przestrzeń możliwych rozwiązań w problemie porządkowania $n$ liczb ma moc $n!$, czyli jest taka sama jak w przypadku problemu komiwojażera. Jednak dzięki odpowiednim algorytmom, $n$ elementów można uporządkować w czasie proporcjonalnym do $n \log n$ lub $n^2$. Więcej na temat porządkowania piszemy w punkcie 4.5.

Dysponując uporządkowanym zbiorem danych, znalezienie w nim elementu lub umieszczenie nowego z zachowaniem porządku jest znacznie łatwiejsze i szybsze, niż gdyby zbiór nie był uporządkowany – piszemy o tym w punkcie 4.4.2.

## 4 PROSTE PROBLEMY I NAJLEPSZE ALGORYTMY ICH ROZWIĄZYWANIA

W tym rozdziale przedstawiamy kilka klasycznych problemów algorytmicznych wraz z możliwie najlepszymi algorytmami ich rozwiązywania. Problemy są rzeczywiście bardzo proste, ale mają bardzo ważne znaczenie w informatyce. Po pierwsze, są rozwiązywane bardzo często w wielu bardziej złożonych sytuacjach problemowych, zarówno z użyciem komputera jak i bez jego pomocy. Po drugie, te problemy stanowią elementy składowe wielu innych metod obliczeniowych i są wywoływane wielokrotnie, dlatego im szybciej potrafimy je rozwiązywać, tym szybciej działają metody rozwiązywania wielu innych problemów.

### 4.1 ZNAJDOWANIE ELEMENTU W ZBIORZE – ZNAJDOWANIE MINIMUM

Zajmiemy się bardzo prostym problemem, który każdy z Was rozwiązuje wielokrotnie w ciągu dnia. Chodzi o znajdowanie w zbiorze elementu, który ma określoną własność. Oto przykładowe sytuacje problemowe:

- znajdź najwyższego ucznia w swojej klasie; a jak zmieni się Twój algorytm, jeśli chciałbyś znaleźć w klasie najniższego ucznia?

- znajdź w swojej klasie ucznia, któremu droga do szkoły zabiera najwięcej czasu;
- znajdź najstarszego ucznia w swojej szkole;
- znajdź największą kartę w potasowanej talii kart;
- wyłonić najlepszego tenisistę w swojej klasie.

Postawiony problem może wydać się zbyt prosty, by zajmować się nim na informatyce – każdy uczeń zapewne potrafi wskazać metodę rozwiązywania, polegającą na systematycznym przeszukaniu całego zbioru danych. Tak pojawia się metoda **przeszukiwania** ciągu, którą można nazwać przeszukiwaniem **liniowym**. Przy tej okazji w dyskusji pojawi się zapewne również **metoda pucharowa**, która jest często stosowana w rozgrywkach turniejowych.

Poczyńmy pewne założenia, które wynikają zarówno z praktycznych sytuacji, jak i odpowiadają stosowanym metodom rozwiązywania. Po pierwsze zakładamy, że przeszukiwane zbiory elementów nie są uporządkowane, np. klasa – od najwyższego do najniższego ucznia lub odwrotnie, gdyż, gdyby tak było, to rozwiązywanie wymienionych wyżej problemów i im podobnych byłoby dziecinnie łatwe – wystarczyłoby wziąć element z początku albo z końca takiego uporządkowania. Po drugie – przyjmujemy także, że nie interesują nas algorytmy rozwiązywania przedstawionych sytuacji problemowych, które w pierwszym kroku porządkują zbiór przeszukiwany, a następnie już prosto znajdują poszukiwane elementy – to założenie wynika z faktu, że sortowanie ciągu jest znacznie bardziej pracochłonne niż znajdowanie wyróżnionych elementów w ciągu.

Z powyższych założeń wynika dość naturalny wniosek, że aby znaleźć w zbiorze poszukiwany element musimy przejrzeć wszystkie elementy zbioru, gdyż jakikolwiek pominięty element mógłby okazać się tym szukanym elementem.

Przy projektowaniu algorytmów istotne jest również określenie, jakie działania (operacje) mogą być wykonywane w algorytmie. W przypadku problemu poszukiwania szczególnego elementu w zbiorze wystarczy, jeśli będziemy umieli porównać elementy między sobą. Przy porównywaniu kart należy uwzględnić ich kolory i wartości, natomiast przy wyłanianiu najlepszego gracza w tenisa, porównania między graczami dokonuje się na podstawie wyniku meczu między nimi.

### Specyfikacja problemu, specyfikacja i opis i algorytm

Dla uproszczenia rozważań zakładamy, że danych jest $n$ liczb w postaci ciąg: $x_1, x_2, \ldots, x_n$. Oto **specyfikacja** rozważanego problemu:

**Problem Min – Znajdowanie najmniejszego elementu w zbiorze**
*Dane:* Liczba naturalna $n$ i zbiór $n$ liczb, dany w postaci ciągu $x_1, x_2, \ldots, x_n$.
*Wynik:* Najmniejsza spośród liczb $x_1, x_2, \ldots, x_n$ – oznaczmy jej wartość przez *min*.

### Algorytm Min

Naszkicowany wyżej algorytm, polegający na przejrzeniu ciągu danych od początku do końca, można zapisać w następujący sposób w postać **listy kroków**:

**Algorytm Min – znajdowanie najmniejszego elementu w zbiorze**
*Krok 1*. Przyjmij za *min* pierwszy element w ciągu, czyli przypisz $min := x_1$.
*Krok 2*. Dla kolejnych elementów $x_i$, gdzie $i = 2, 3, \ldots, n$, jeśli *min* jest większe niż $x_i$, to za *min* przyjmij $x_i$, czyli, jeśli $min > x_i$, to przypisz $min := x_i$.

Metoda zastosowana w algorytmie **Min**, polegająca na badaniu elementów ciągu danych w kolejności, w jakiej są ustawione, nazywa się **przeszukiwaniem liniowym**.

Algorytm **Min** może być łatwo zmodyfikowany tak, aby otrzymać algorytm **Max**, służący do znajdowania największego elementu w ciągu – wystarczy w tym celu zmienić tylko zwrot nierówności w kroku 2.

Często, poza znalezieniem elementu najmniejszego (lub największego) chcielibyśmy znać jego położnie, czyli miejsce (numer) w ciągu danych. W tym celu wystarczy wprowadź nową zmienną, np. *imin*, w której będzie przechowywany numer aktualnie najmniejszego elementu.

W tych materiałach poprzestajemy na opisach algorytmów w postaci listy kroków. Inną reprezentacją algorytmu jest **schemat blokowy**. Najbardziej precyzyjną postać ma **implementacja** algorytmu w postaci programu w języku programowania. Działanie algorytmu można również zademonstrować posługując się programem edukacyjnym. W materiałach do tego wykładu są udostępnione programy **Maszyna sortująca** i **Sortowanie**, które mogą być wykorzystane do eksperymentów z algorytmem **Min** i z algorytmami sortowania (patrz punkt 4.5).

**Pracochłonność (złożoność) algorytmu Min**

W algorytmie **Min** podstawową operacją jest porównanie dwóch elementów ze zbioru danych – policzmy więc, ile porównań jest wykonywanych w tym algorytmie. W każdej iteracji algorytmu jest wykonywane jedno porównanie $min > x_i$, a zatem w całym algorytmie jest wykonywanych $n - 1$ porównań. Pozostałe operacje służą głównie do organizacji obliczeń i ich liczba jest związana z liczbą porównań. Na przykład, operacja przypisania $min := x_i$ może być wykonana tylko o jeden raz więcej – w kroku 1 i $n - 1$ razy w kroku 2. Możemy więc podsumować nasze rozumowanie bardzo ważnym stwierdzeniem

> najmniejszy (lub największy) element w niepustym zbiorze danych można znaleźć wykonując o jedno porównanie mniej niż wynosi liczba wszystkich elementów w tym zbiorze.

Zastanówmy się teraz, czy w zbiorze złożonym z $n$ liczb, można znaleźć najmniejszy element wykonując mniej niż $n - 1$ porównań elementów tego zbioru? Udzielimy negatywnej odpowiedzi na to pytanie[1], posługując się interpretacją wziętą z klasowego turnieju tenisa. Ile należy rozegrać meczów (to są właśnie porównania w przypadku tego problemu), aby wyłonić najlepszego tenisistę w klasie? Lub inaczej – kiedy możemy powiedzieć, że Janek jest w naszej klasie najlepszym tenisistą? Musimy mieć pewność, że wszyscy pozostali uczniowie są od niego gorsi, czyli przegrali z nim, bezpośrednio lub pośrednio. A zatem każdy inny uczeń przegrał przynajmniej jeden mecz, czyli rozegranych zostało przynajmniej tyle meczów, ilu jest uczniów w klasie mniej jeden. I to kończy nasze uzasadnienie.

Z dotychczasowych rozważań wynika, że algorytm **Min** jest najlepszym algorytmem służącym do znajdowania najmniejszego elementu, gdyż wykonywanych jest w nim tyle porównań, ile musi wykonać jakikolwiek algorytm rozwiązywania tego problemu. O takim algorytmie mówimy, że jest **algorytmem optymalnym pod względem złożoności obliczeniowej**.

## 4.2 KOMPLETOWANIE PODIUM ZWYCIĘZCÓW TURNIEJU

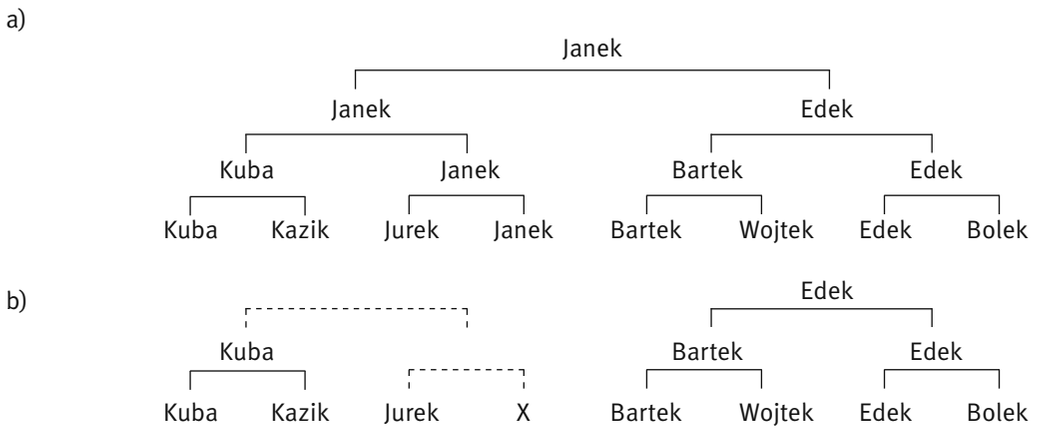
Przedstawiony w poprzednim punkcie algorytm nie jest jedyną metodą służącą do znajdowania najlepszego elementu w zbiorze. Inną metodą jest tzw. **system pucharowy**, stosowany często przy wyłanianiu najlepszego zawodnika bądź drużyny w turnieju. W metodzie tej „porównanie" dwóch zawodników (lub drużyn), by stwierdzić, który jest lepszy („większy"), polega na rozegraniu meczu, który kończy się zwycięstwem jednego z zawodników.

**Wyłanianie zwycięzcy w turnieju**

Nurtować może pytanie, czy znajdowanie najlepszego zawodnika systemem pucharowym nie jest czasem metodą bardziej efektywną pod względem liczby wykonywanych porównań (czyli rozegranych meczy), niż przeszukiwanie liniowe, opisane w poprzednim rozdziale. Na rysunku 2(a) jest przedstawiony fragment turnieju, rozegranego między ośmioma zawodnikami. Zwycięzcą okazał się Janek po rozegraniu w całym turnieju siedmiu meczów. A zatem podobnie jak w przypadku metody liniowej, aby wyłonić zwycięzcę, czyli najlepszego zawodnika (elementu) wśród ośmiu zawodników, należało rozegrać o jeden mecz mnie niż wystąpiło w turnieju zawodników. Nie jest to przypadek.

Powyższa prawidłowość wynika z następującego faktu: schemat turnieju jest drzewem binarnym, a w takim drzewie liczba wierzchołków pośrednich jest o jeden mniejsza od liczby wierzchołków końcowych. Wierzchołki końcowe to zawodnicy przystępujący do turnieju, a wierzchołki pośrednie odpowiadają rozegranym meczom.

[1] Posługujemy się tutaj argumentacją zaczerpniętą z książki Hugona Steinhausa, *Kalejdoskop matematyczny* (WSiP, Warszawa 1989, rozdz. III), [5].




*Rysunek 2.*
Drzewo przykładowych rozgrywek w turnieju tenisowym (a) oraz drzewo znajdowania drugiego najlepszego zawodnika turnieju (b)

### Wyłanianie drugiego najlepszego zawodnika turnieju

Bardzo ciekawy problem postawił około 1930 roku Hugo Steinhaus: jaka jest najmniejsza liczba meczów tenisowych potrzebnych do wyłonienia najlepszego i drugiego najlepszego zawodnika turnieju. W turniejach drugą nagrodę otrzymuje zwykle zawodnik pokonany w finale. I tutaj Steinhaus miał słuszne wątpliwości, czy jest to właściwa decyzja, tzn., czy pokonany w finale jest drugim najlepszym zawodnikiem turnieju, czyli czy jest lepszy od wszystkich pozostałych zawodników z wyjątkiem zwycięzcy turnieju. Wątpliwości H. Steinhausa były uzasadnione – spójrzmy na drzewo turnieju przedstawione na rysunek 2(a). Zwycięzcą w tym turnieju jest Janek, który w finale pokonał Edka. Edkowi przyznano więc drugą nagrodę, chociaż wykazał, że jest lepszy jedynie od Bolka, Bartka i Wojtka (gdyż przegrał z Bartkiem). Nic nie wiemy, jakby Edek grał przeciwko zawodnikom z poddrzewa, z którego jako zwycięzca został wyłoniony Janek. Jak można naprawić ten błąd organizatorów rozgrywek tenisowych? Istnieje prosty sposób znalezienia drugiego najlepszego zawodnika turnieju – rozegrać jeszcze jedną pełną rundę z pominięciem zwycięzcy turnieju głównego. Wówczas, najlepszy i drugi najlepszy zawodnik zawodów zostaliby wyłonieni w $2n–3$ meczach. Steinhaus oczywiście znał to rozwiązanie, ale pytał o najmniejszą potrzebną liczbę meczów.

Jeśli chcemy, aby drugi najlepszy zawodnik nie musiał być wyłaniany w nowym pełnym turnieju, to musimy umieć skorzystać ze wszystkich wyników głównego turnieju. Posłużymy się drzewem turnieju z rysunku 2(a). Zauważmy, że Edek jest oczywiście najlepszy wśród zawodników, którzy w drzewie rozgrywek znajdują się w wierzchołkach leżących poniżej najwyższego wierzchołka, który on zajmuje. Musimy więc jedynie porównać go z zawodnikami drugiego poddrzewa. Aby i w tym poddrzewie wykorzystać wyniki dotychczasowych meczów, eliminujemy z niego Janka — zwycięzcę turnieju i wstawiamy Edka na jego początkowe miejsce X. Spowoduje to, że Edek zostanie porównany z najlepszymi zawodnikami w drugim poddrzewie. Na rysunku 2(b) oznaczyliśmy przerywana linią mecze, które zostaną rozegrane w tej części turnieju – Jurek z Edkiem i zwycięzca tego meczu z Kubą, a więc dwa dodatkowe mecze.

Algorytm ten można, po zmianie słownictwa, zastosować do znajdowania największej i drugiej największej liczby w zbiorze danych.

### Złożoność wyłaniania zwycięzcy i drugiego najlepszego zawodnika turnieju

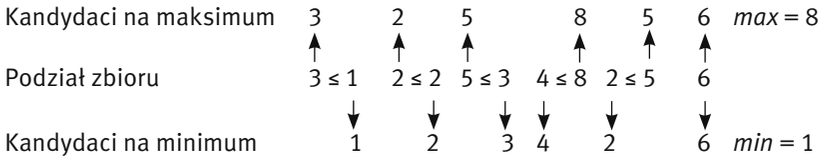
Ile porównań jest wykonywanych w opisanym algorytmie znajdowania najlepszego i drugiego najlepszego zawodnika w turnieju? Najlepszy zawodnik jest wyłaniany w $n - 1$ meczach, gdzie $n$ jest liczbą wszystkich zawodników. Z kolei, aby wyłonić drugiego najlepszego zawodnika, trzeba rozegrać tyle meczów, ile jest poziomów w drzewie turnieju głównego (z wyjątkiem pierwszego poziomu). Dla uproszczenia przyjmijmy, że drzewo jest **pełne**, tzn. każdy zawodnik ma parę, czyli w każdej rundzie turnieju gra parzysta liczba zawodników. Stąd wynika, że na najwyższym poziomie jest jeden zawodnik, na poziomie niższym – dwóch, na kolejnym – czterech itd. Czyli, liczba zawodników rozpoczynających turniej jest potęgą liczby 2, zatem $n = 2^k$, gdzie $k$ jest liczbą poziomów drzewa – oznaczmy ją przez $\log_2 n$. Algorytm wykonuje więc $(n - 1) + (\log_2 n - 1) = n + \log_2 n - 2$ porównań. Jeśli $n$ nie jest potęgą liczby 2, to na ogół w turnieju niektórzy zawodnicy otrzymują wolną kartę, a podana liczba jest oszacowaniem z góry liczby rozegranych meczów.

Przedstawiony powyżej algorytm znajdowania najlepszego i drugiego najlepszego zawodnika turnieju jest również optymalny, tzn. jest najszybszy w sensie liczby rozegranych meczów (porównań).

## 4.3 JEDNOCZESNE ZNAJDOWANIE NAJMNIEJSZEGO I NAJWIĘKSZEGO ELEMENTU

Jedną z miar, określającą, jak bardzo są porozrzucane wartości obserwowanej w doświadczeniu wielkości, jest **rozpiętość** zbioru, czyli różnica między największą (w skrócie, maksimum) a najmniejszą wartością elementu (w skrócie, minimum) w zbiorze. Interesujące jest więc jednoczesne znalezienie najmniejszej i największej wartości w zbiorze liczb. Na podstawie dotychczasowych rozważań, dotyczących wyznaczania najmniejszej i największej wartości w zbiorze liczb, łatwo można podać algorytm znajdowania jednocześnie obu tych elementów w zbiorze. W tym celu stosujemy najpierw algorytm **Min** do całego zbioru, a później algorytm **Max** do zbioru z usuniętym minimum. W takim algorytmie „jednoczesnego wyznaczania" minimum i maksimum w ciągu złożonym z $n$ liczb jest wykonywanych $(n-1)+(n-2)=2n-3$ porównań. Ale czy rzeczywiście te dwie wielkości są wyznaczane jednocześnie?

Postaramy się znacznie przyspieszyć to postępowanie, a będzie to polegało na rzeczywiście jednoczesnym szukaniu najmniejszego i największego elementu w całym zbiorze. W tym celu zauważmy, że jeśli dwie liczby $x$ i $y$ spełniają nierówność $x \le y$, to $x$ jest kandydatem na najmniejszą liczbę w zbiorze, a $y$ jest kandydatem na największą liczbę w zbiorze. (Jeśli prawdziwa jest nierówność przeciwna, to wnioskujemy odwrotnie.) A zatem, porównując elementy parami, można podzielić dany zbiór elementów na dwa podzbiory, kandydatów na minimum i kandydatów na maksimum, i w tych zbiorach – które są niemal o połowę mniejsze niż oryginalny zbiór! – szukać odpowiednio minimum i maksimum. Gdy zbiór ma nieparzystą liczbę elementów, to ostatni element ciągu dodajemy do jednego i do drugiego podzbioru kandydatów. Postępowanie to jest zilustrowane przykładem na rys. 3, a opis algorytmu pozostawiamy do samodzielnego wykonania.




*Rysunek 3.*
Przykład postępowania podczas jednoczesnego znajdowania minimum i maksimum w ciągu liczb

Naszkicowany algorytm jest przykładem metody, leżącej u podstaw bardzo wielu efektywnych algorytmów. Można w nim wyróżnić dwa etapy:

- podziału danych na dwa podzbiory (kandydatów na minimum i kandydatów na maksimum);
- zastosowanie znanych algorytmów **Min** i **Max** do utworzonych podzbiorów danych.

Jest to przykład zasady (metody) **dziel i zwyciężaj**, jednej z najefektywniejszych metod algorytmicznych w informatyce – patrz punkty 4.4.2 i 4.5.2. **Dziel** – odnosi się do podziału zbioru danych na podzbiory, zwykle o jednakowej liczbie elementów, do których następnie są stosowane odpowiednie algorytmy. **Zwycięstwo** – to efekt końcowy, czyli efektywne rozwiązanie rozważanego problemu.

Obliczmy, ile porównań między elementami danych jest wykonywanych w powyższym algorytmie. Gdy $n$ jest liczbą parzystą, to w kroku podziału jest wykonywanych $n/2$ porównań, a znalezienie *min* oraz znalezienie *max* wymaga każde $n/2-1$ porównań. Razem jest to $3n/2-2$ porównania. Gdy $n$ jest liczbą nieparzystą, to otrzymujemy liczbę porównań $\lceil 3n/2 \rceil - 2$, gdzie $\lceil x \rceil$ oznacza tzw. **powałę liczby**, czyli najmniejszą liczbę całkowitą $k$ spełniającą nierówność $x \le k$. A zatem, w tym algorytmie jest wykonywanych $\lceil 3n/2 \rceil - 2$ porównania, czyli ok. $n/2$ mniej porównań niż w algorytmie naiwnym, naszkicowanym na początku tego punktu. Dodajmy, że jest to również algorytm optymalny pod względem liczby wykonywanych porównań.

## 4.4 POSZUKIWANIE ELEMENTÓW W ZBIORZE

Komputery ułatwiają szybkie poszukiwanie informacji umieszczonych na płytach CD i na serwerach sieci Internet. Jest to zasługą szybkości działania procesorów, jak również dobrej organizacji pracy, dzięki czemu pozostaje im ... niewiele do roboty.

W punkcie 4.1. zajmowaliśmy się znajdowaniem szczególnych elementów w zbiorze nieuporządkowanym, najmniejszego i największego. Teraz będziemy rozważać problem poszukiwania w ogólniejszej postaci.

**Problem poszukiwania elementu w zbiorze**
*Dane:* Zbiór elementów w postaci ciągu $n$ liczb $x_1, x_2, ..., x_n$. Wyróżniony element $y$.
*Wynik:* Jeśli $y$ należy do tego zbioru, to podaj jego miejsce (indeks) w ciągu, a w przeciwnym razie – sygnalizuj brak takiego elementu w zbiorze.

Problem poszukiwania ma bardzo wiele zastosowań i jest rozwiązywany przez komputer na przykład wtedy, gdy w jakimś ustalonym zbiorze informacji staramy się znaleźć konkretną informację. Rozważana przez nas wersja tego problemu jest bardzo prosta, na ogół bowiem zbiory i ich elementy mają bardzo złożoną postać, nie są ograniczone tylko do pojedynczych liczb. Przedstawione przez nas metody mogą być jednak uogólnione na bardziej złożone sytuacje problemowe.

W następnych podpunktach, najpierw rozwiązujemy problem poszukiwania w dowolnym zbiorze elementów, a później – w zbiorze uporządkowanym.

### 4.4.1 POSZUKIWANIE ELEMENTU W ZBIORZE NIEUPORZĄDKOWANYM

Jeśli nic nie wiemy o elementach w ciągu danych $x_1, x_2, ..., x_n$, to aby stwierdzić, czy wśród nich jest element równy danemu $y$, musimy sprawdzić każdy z elementów tego ciągu, gdyż element $y$ może się znajdować w dowolnym miejscu ciągu, a w szczególności może go tam nie być. W takim przypadku stosujemy **przeszukiwanie** (lub **poszukiwanie**) **liniowe**, które stosowaliśmy w punkcie 4.1 do znajdowania w ciągu elementu najmniejszego lub największego. Na ogół takie przeszukiwanie odbywa się „od lewej do prawej", czyli od początku do końca ciągu. Można je opisać następująco.

**Algorytm poszukiwania liniowego** (dla specyfikacji powyżej)
*Krok 1.* Dla $i = 1, 2, ..., n$, jeśli $x_i = y$, to przejdź do kroku 3.
*Krok 2.* Komunikat: W ciągu danych nie ma elementu równego $y$. Zakończ algorytm.
*Krok 3.* Element równy $y$ znajduje się na miejscu $i$ w ciągu danych. Zakończ algorytm.

Jeśli element $y$ znajduje się w przeszukiwanym ciągu, to algorytm kończy działanie po natknięciu się na niego po raz pierwszy, a jeśli nie ma go w tym ciągu to kończy się po dojściu do końca ciągu. W obu przypadkach liczba działań jest proporcjonalna do liczby elementów w ciągu. W pierwszym przypadku najwięcej operacji jest wykonywanych wówczas, gdy poszukiwany element jest na końcu ciągu.

**Przeszukiwanie liniowe z wartownikiem**
Ciekawe własności ma niewielka modyfikacja powyższego algorytmu, wykorzystująca specjalny element, umieszczony na końcu ciągu, zwany **wartownikiem**. Rolą wartownika jest „pilnowanie", by proces przeszukiwania nie wyszedł poza ciąg. Jak wiemy, gdy ciąg zawiera element o wartości $y$, to przeszukiwanie kończy się na tym elemencie. Aby mieć pewność, że przeszukiwanie zawsze zakończy się na elemencie o wartości $y$, dołączamy na końcu ciągu element o wartości $y$. W efekcie, przeszukiwanie zawsze zakończy się znalezieniem elementu o wartości $y$, należy jedynie sprawdzić, czy ten element znajduje się na dołączonej pozycji zbioru, czy też wystąpił wcześniej. W pierwszym przypadku, badany zbiór nie zawiera elementu równego $y$, a w drugim – $y$ należy do zbioru. Widać stąd, że dołączony do zbioru element odgrywa rolę jego wartownika – nie musimy bowiem sprawdzać, czy przeglądanie objęło cały zbiór czy nie – zawsze zatrzyma się ono na szukanym elemencie, którym może być dołączony właśnie element.

### 4.4.2 POSZUKIWANIE ELEMENTU W ZBIORZE UPORZĄDKOWANYM

W tym punkcie zakładamy, że poszukiwania elementów (informacji) są prowadzone w uporządkowanych ciągach elementów – chcemy albo znaleźć element, albo umieścić go w takim ciągu z zachowaniem uporządkowania.

**Porządek w informacjach**
Zbiory mogą mieć różną strukturę – mogą to być książki w bibliotece, hasła w encyklopedii, liczba w ustalonym przedziale lub numery w książce telefonicznej. Te przykłady są bliskie codziennym sytuacjom, w których

należy odszukać pewną informację i zapewne stosowane przez Was w tych przypadkach metody są podobne do opisanych tutaj. Naszymi rozważaniami chcemy utwierdzić Was w przekonaniu, że:

**integralną częścią informacji jest jej uporządkowanie,**

gdyż w przeciwnym razie ... nie jest to informacja. To stwierdzenie nie jest naukowym określeniem informacji[2], ale odnosi się do informacji w potocznym znaczeniu.

Wykonaj teraz ćwiczenie, które zapewne wykonywałeś już nieraz w swoim życiu, nie zdając sobie nawet z tego sprawy. Weź do ręki jedną z książek: słownik ortograficzny, słownik polsko-angielski lub książkę telefoniczną, wybierz trzy słowa zaczynające się na litery: *c*, *k* oraz *w* i znajdź je w wybranej książce. Zanotuj, ile razy ją otwierałaś, zanim znalazłeś stronę z poszukiwanym słowem.

Jeśli książka, którą wybrałeś, ma między 1000 a 2000 stron, to dla znalezienia jednego słowa nie powinieneś otwierać jej częściej niż 11 razy; jeśli ma między 500 a 1000 stron – to nie częściej niż 10 razy; jeśli między 250 a 500 stron – to nie częściej niż 9 razy itp.

Skąd to wiemy? Przypuszczamy, że w poszukiwaniu hasła, po zajrzeniu na wybraną stronę wiesz, że znajduje się ono przed nią, albo po niej, możesz więc jedną z części książki pominąć w dalszych poszukiwaniach. Co więcej, w nieodrzuconej części kartek wybierasz jako kolejną tę, która jest bliska środka, lub leży w pobliżu litery, na którą zaczyna się poszukiwany wyraz. Stosujesz więc – może nawet o tym nie wiedząc – metodę poszukiwania, która polega na **podziale (połowieniu) przeszukiwanego zbioru**. Możesz ją zastosować, bo przeszukiwany zbiór jest uporządkowany. A ile prób musiałbyś wykonać, gdyby hasła w słowniku nie były uporządkowane? Porównaj teraz:

- W alfabetycznym spisie telefonów na 1000 stronach wystarczy przejrzeć co najwyżej 10 stron, by znaleźć numer telefonu danej osoby.
- A jeśli miałbyś znaleźć osobę, która ma telefon o numerze 1234567, to w najgorszym przypadku musiałbyś przejrzeć wszystkie 1000 stron!

Czy to porównanie nie świadczy o potędze uporządkowania i o sile algorytmu zastosowanego do uporządkowanego wykazu?

Przedstawiony sposób poszukiwania przez połowienie w zbiorze uporządkowanym ilustrują, że ta metoda jest kolejnym zastosowaniem **zasady dziel i zwyciężaj**.

### Algorytm poszukiwania przez połowienie

Algorytm poszukiwania przez połowienie jest zwany również **binarnym poszukiwaniem**. Przyjmijmy, że przeszukiwany ciąg liczb jest umieszczony w tablicy $x[k..l]$ i załóżmy dla uproszczenia, że wartość poszukiwanego elementu $y$ mieści się w przedziale wartości elementów w tej tablicy, czyli $x_k \le y \le x_l$. Algorytm, który podajemy gwarantuje, że w trakcie jego działania przeszukiwany przedział zawiera poszukiwany element $y$, czyli $x_{lewy} \le y \le x_{prawy}$. Ta własność oraz to, że długość tego przedziału zmniejsza się w każdej iteracji (zob. krok 3), zapewniają, że poniższy algorytm jest poprawny i skończony.

**Algorytm poszukiwania przez połowienie (algorytm binarnego przeszukiwania)**

*Dane:* Uporządkowany ciąg liczb w tablicy $x[k..l]$, tzn. $\boldsymbol{x_k \le x_{k+1} \le \ldots \le x_l}$; oraz element $y$ spełniający nierówności $\boldsymbol{x_k \le y \le x_l}$.

*Wyniki:* Takie $s$ ($k \le s \le l$), że $x_s = y$, lub przyjąć $s = -1$, jeśli $y \ne x_i$ dla każdego $i$ ($k \le i \le l$).

*Krok 1.* *lewy* := *k*; *prawy* := *l*; {Początkowe końce przeszukiwanego przedziału.}

*Krok 2.* Jeśli *lewy* › *prawy*, to przypisz *s* := –1 i zakończ algorytm.
{Oznacza to, że poszukiwanego elementu *y* nie ma w przeszukiwanej tablicy.}

*Krok 3.* *s*:=(*lewy* + *prawy*) div 2; {Operacja div oznacza dzielenie całkowite.}
Jeśli $x_s = y$, to zakończ algorytm. {Znaleziono element *y* w przeszukiwanej tablicy.}
Jeśli $x_s \lessdot y$, to *lewy* :=*s* + 1, a w przeciwnym razie *prawy* := *s* – 1.
Wróć do kroku 2.

[2] W teorii informacji, informacja jest definiowana jako „miara niepewności zajścia pewnego zdarzenia spośród skończonego zbioru zdarzeń możliwych" – na podstawie *Nowej encyklopedii powszechnej PWN*.

**Złożoność algorytmu binarnego przeszukiwania**

Pytanie o liczbę porównań w powyższym algorytmie można sformułować następująco: ile razy należy odrzucać połowę bieżącego ciągu, by pozostał tylko jeden element (zauważmy tutaj, że jeśli elementu *y* nie ma w ciągu, to kontynuujemy algorytm aż do wyczerpania wszystkich elementów, czyli wykonujemy o jeden krok więcej). Jeśli $n = 32$, to jeden element pozostaje po pięciu podziałach, a jeśli $n = 16$ – to po czterech. Stąd można wywnioskować, że jeśli wartość *n* zawiera się między 16 a 32, to wykonujemy nie więcej niż pięć porównań. Ta obserwacja ma związek z potęgą liczby 2, a dokładniej – z najmniejszym wykładnikiem *k* potęgi $2^k$, której wartość nie jest mniejsza od *n*, czyli $n \leq 2^k$. Pojawia się więc tutaj w naturalny sposób funkcja odwrotna do potęgowania – **logarytm**. Można nawet przyjąć **„informatyczną" definicję tej funkcji**:

> $\log_2 n$ jest równy liczbie kroków prowadzących od *n* do 1, w których bieżąca liczba jest zastępowana przez zaokrąglenie w górę jej połowy.

**Algorytm binarnego umieszczania**

Algorytm binarnego przeszukiwania ma dość istotne uogólnienie, gdy dla elementu *y*, bez względu na to, czy należy do ciągu czy nie, chcemy znaleźć takie miejsce, by po wstawieniu go tam, ciąg pozostał uporządkowany. Odpowiedni algorytm można w tym przypadku nazwać **binarnym umieszczaniem** i jest on prostym rozszerzeniem powyższego algorytmu.

**Poszukiwanie interpolacyjne, czyli poszukiwania w słownikach**

Czy rzeczywiście przeszukiwanie binarne jest najszybszą metodą znajdowania elementu w ciągu uporządkowanym? Wyobraźmy sobie, że mamy znaleźć w książce telefonicznej numer telefonu Pana Bogusza Alfreda. Wtedy zapewne skorzystamy z tego, że litera B jest blisko początku alfabetu i, owszem, zastosujemy metodę podziału, ale w pierwszej próbie nie będziemy jednak dzielić książki na dwie połowy, ale raczej spróbujemy trafić blisko tych stron, na których znajdują się nazwiska zaczynające się na literę B. W dalszych krokach będziemy postępować podobnie. Tę obserwację można wykorzystać w algorytmie poszukiwania. Zauważmy najpierw, że w algorytmach binarnych jest sprawdzana jedynie relacja, **czy** dana liczba *y* jest większa (lub mniejsza lub równa) od wybranej z ciągu, natomiast nie sprawdzamy i nie wykorzystujemy tego, **jak bardzo** jest większa. Podczas odnajdywania wyrazów w encyklopediach korzystamy natomiast z informacji, w jakim miejscu alfabetu znajduje się litera, którą rozpoczyna się poszukiwany wyraz, i w zależności od tego wybieramy odpowiednią porcję kartek. Strategia ta nazywa się **interpolacyjnym poszukiwaniem**, gdyż uwzględnia nie tylko położeniu szukanej liczby względem środka ciągu, ale uwzględnia jej wartość względem rozpiętości krańcowych wartości w ciągu. Szczegółowe informacje na temat interpolacyjnego poszukiwania można znaleźć w książce [6].

## 4.5 ALGORYTMY PORZĄDKOWANIA

**Porządkowanie**, nazywane również często **sortowaniem**, ma olbrzymie znaczenie niemal w każdej działalności człowieka. Jeśli elementy w zbiorze są uporządkowane zgodnie z jakąś regułą (np. książki lub ich karty katalogowe według liter alfabetu, słowa w encyklopedii, daty, numery telefonów według nazwisk właścicieli), to wykonywanie wielu operacji na tym zbiorze staje się znacznie łatwiejsze i szybsze. Między innymi dotyczy to operacji (patrz punkt 4.4):

- sprawdzenia czy dany element, czyli element o ustalonej wartości cechy, według której zbiór został uporządkowany, znajduje się w zbiorze,
- znalezienia elementu w zbiorze, jeśli w nim jest,
- dołączenia nowego elementu w odpowiednie miejsce, aby zbiór pozostał nadal uporządkowany.

Komputery w dużym stopniu zawdzięczają swoją szybkość temu, że działają na uporządkowanych informacjach. To samo odnosi się do nas, gdy posługujemy się informacjami i komputerami. Jeśli chcemy na przykład sprawdzić, czy w jakimś katalogu dyskowym znajduje się plik o podanej nazwie, rozszerzeniu, czasie utworzenia lub rozmiarze, to najpierw odpowiednio porządkujemy listę plików i wtedy na ogół znajdujemy odpowiedź natychmiast.

**Problem porządkowania**

Dla uproszczenia, problem porządkowania sformułujemy dla liczb, chociaż wiele praktycznych problemów może dotyczyć porządkowania innych obiektów przechowywanych w komputerze.

**Problem porządkowania (sortowania)**
*Dane:* Liczba naturalna $n$ i ciąg $n$ liczb $x_1, x_2, ..., x_n$
*Wynik:* Uporządkowanie tego ciągu liczb od najmniejszej do największej.

Z założenia, że porządkujemy tylko liczby względem ich wartości wynika, że interesują nas algorytmy, w których główną operacją jest porównanie, wykonywane między elementami danych. W ogólnym sformułowaniu problemu porządkowania, porządkowane obiekty mogą być bardziej złożone, np. słowa (przy tworzeniu słownika), adresy (do książki telefonicznej) czy bardzo złożone zestawy danych, jak np. informacje o koncie bankowym i jego właścicielu.

Znanych jest bardzo wiele algorytmów porządkowania liczb i innych obiektów przechowywanych w komputerze. Temu zagadnieniu poświęcono wiele opasłych książek, np. Donald Knuth napisał na ten temat ponad 1000 stron jeszcze w latach 60. XX wieku, patrz [4]. W dalszej części tego punktu prezentujemy dwa najbardziej znane algorytmy porządkowania – przez wybór oraz sortowanie przez scalanie. Pierwszy z nich jest iteracją poznanego wcześniej algorytmu i ma złożoność proporcjonalną do $n^2$, gdzie $n$ jest liczbą porządkowanych elementów, a drugi – to przykład metody dziel i zwyciężaj, będący jedną z najszybszych metod sortowania. Złożoność tego drugiego algorytmu wynosi $n\log n$.

Działanie prezentowanych algorytmów porządkowania, jak i wielu innych algorytmów sortujących można obejrzeć w programach edukacyjnych **Maszyna sortująca** i **Sortowanie**, które znajdują się w folderze z materiałami do tych zajęć.

### 4.5.1 PORZĄDKOWANIE PRZEZ WYBÓR

Jeden z najprostszych algorytmów porządkowania można wyprowadzić korzystając z tego, co już poznaliśmy w poprzednich punktach. Zauważmy, że jeśli mamy ustawić elementy w kolejności od najmniejszego do największego, to najmniejszy element w zbiorze powinien się znaleźć na początku tworzonego ciągu, za nim powinien być umieszczony najmniejszy element w zbiorze pozostałym po usunięciu najmniejszego elementu itd. Taki algorytm jest więc **iteracją** znanego algorytmu znajdowania **Min** w ciągu i nosi nazwę **algorytmu porządkowania przez wybór**. Oto jego opis:

**Algorytm porządkowani przez wybór – SelectionSort**
*Dane:* Liczba naturalna $n$ i ciąg $n$ liczb $x_1, x_2, ..., x_n$.
*Wynik:* Uporządkowanie danego ciągu liczb od najmniejszej do największej. (*Uwaga*. Elementy ciągu danego i wynikowego oznaczamy tak samo, gdyż porządkowanie odbywa się „w tym samym miejscu”.)
*Krok 1*. Dla $i = 1, 2, ..., n - 1$ wykonaj kroki 2 i 3, a następnie zakończ algorytm.
*Krok 2*. Znajdź $k$ takie, że $x_k$ jest najmniejszym elementem w ciągu $x_i, ..., x_n$.
*Krok 3*. Zamień miejscami elementy $x_i$ oraz $x_k$.

**Złożoność algorytmu SelectionSort**
Aby obliczyć, ile porównań i zamian elementów, w zależności od liczby elementów w ciągu $n$, jest wykonywanych w algorytmie **SelectionSort** wystarczy zauważyć, że ten algorytm jest iteracją algorytmu znajdowania najmniejszego elementu w ciągu, a ciąg, w którym szukamy najmniejszego elementu, jest w kolejnych iteracjach coraz krótszy. Liczba przestawień elementów jest równa liczbie iteracji, gdyż elementy są przestawiane jedynie na końcu każdej iteracji, których jest $n - 1$, a więc wynosi $n - 1$. Jeśli zaś chodzi o liczbę porównań, to wiemy, że algorytm znajdowania minimum w ciągu wykonuje o jedno porównanie mniej niż jest elementów w ciągu, a zatem cały algorytm porządkowania przez wybór, dla ciągu danych złożonego na początku z $n$ elementów wykonuje liczbę porównań równą:

$$(n - 1) + (n - 2) + (n - 3) + ... + 3 + 2 + 1$$

Wartość tej sumy można obliczyć wieloma sposobami, np. jako sumę postępu arytmetycznego i otrzymujemy

$$\frac{(n-1)n}{2}$$

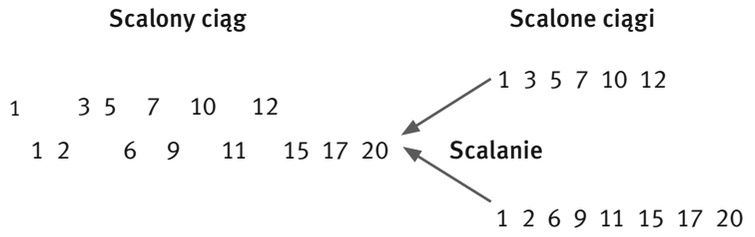
Zauważmy, że otrzymany wzór na liczbę porównań w algorytmie porządkowania przez wybór jest dokładną liczbą działań wykonywanych w algorytmie, bez względu na to, jak daleko porządkowany ciąg jest już uporządkowany.

### 4.5.2 PORZĄDKOWANIE PRZEZ SCALANIE

Metoda **dziel i zwyciężaj** realizowana rekurencyjnie jest jedną z najsilniejszych technik komputerowego rozwiązywania problemów. W tej metodzie, część organizacji obliczeń jest „zrzucana na komputer", faktycznie więc wykorzystywana jest potęga komputerów. W tym punkcie zastosujemy metodę dziel i zwyciężaj w algorytmie porządkowania przez scalanie. W tym algorytmie jest wykorzystywana **metoda scalania** uporządkowanych ciągów, czyli ich łączenia w jeden ciąg.

#### Scalanie ciągów uporządkowanych

Przyjmijmy, że scalamy ciągi uporządkowane i wynik scalania ma być również ciągiem uporządkowanym. Można to wykonać w bardzo prosty sposób: patrzymy na początki danych ciągów i do tworzonego ciągu przenosimy mniejszy z elementów czołowych lub którykolwiek z nich, jeśli są równe, patrz rys. 4.




*Rysunek 4.*
Przykład scalania dwóch ciągów uporządkowanych

Aby określić, ile porównań wykonuje algorytm scalający dwa ciągi zauważmy, że z wyjątkiem elementu przenoszonego na końcu, przeniesienie każdego innego elementu może być związane z wykonaniem porównania między elementami danych ciągów. Tak jest wtedy, gdy w kroku przed przedostatnim, oba scalane ciągi zawierają jeszcze po jednym elemencie. Przypuśćmy więc, że na początku jeden ciąg zawiera $k$ elementów a drugi ma $l$ elementów. razem $n$, czyli $n = k + l$. Ta dyskusja prowadzi do konkluzji, że bez względu na liczności danych ciągów, ich scalenie może wymagać wykonania $n - 1$ porównań.

Z tej dyskusji wynika jeszcze jeden wniosek – ponieważ nie potrafimy przewidzieć, który z ciągów i kiedy wyczerpie się w trakcie scalania, tworzony ciąg nie może być umieszczany na miejscu ciągów danych do scalenia, musi więc być tworzony na nowym miejscu. Zatem potrzebna jest dodatkowa pamięć – mówimy w takim przypadku, że ta metoda nie działa „w miejscu" (*in situ*).

#### Algorytm scalania dwóch ciągów uporządkowanych – Scal

*Dane:* Dwa uporządkowane ciągi *x* i *y*.
*Wynik:* Uporządkowany ciąg *z*, będący scaleniem ciągów *x* i *y*.
*Krok 1.* Dopóki oba ciągi *x* i *y* nie są puste wykonuj następującą operację:
przenieś mniejszy z najmniejszych elementów z ciągów *x* i *y* do ciągu *z*.
*Krok 2.* Do końca ciągu *z* dopisz elementy pozostałe w jednym z ciągów *x* lub *y*.

#### Porządkowanie przez scalanie

Algorytm scalania dwóch uporządkowanych ciągów można stosować do tworzenia uporządkowanych ciągów z podciągów już uporządkowanych. Mógłby być pożytek z tego algorytmu przy porządkowaniu, gdybyśmy potrafili najpierw uporządkować te podciągi. Załóżmy więc, że te podciągi porządkujemy... tą samą metodą – w tym celu przydaje się **rekurencja**. W algorytmie porządkowania przez scalanie porządkowany ciąg jest dzielony w każdym kroku na dwa, niemal równej długości podciągi, które rekurencyjnie są porządkowane tą samą metodą. Warunkiem zakończenia rekurencji w tym algorytmie jest sytuacja, gdy ciąg ma jeden element, wtedy nie można go już podzielić na podciągi, chociaż nie ma nawet po co – jest to bowiem ciąg już uporządkowany. Jak zwykle, w opisie algorytmu rekurencyjnego, po nazwie algorytmu występuje układ parametrów określających rozwiązywany problem, który jest wykorzystany w treści algorytmu, w odwołaniu do niego samego przy rozwiązywaniu mniejszych podproblemów.

**Algorytm porządkowania przez scalanie MergeSort** ($l,p,x$)
*Dane:* Ciąg liczb $x_l, x_{l+1}, ..., x_p$
*Wynik:* Uporządkowanie tego ciągu liczb od najmniejszej do największej.
*Krok 1.* Jeśli $l < p$, to przyjmij $s:=(l+p)$ div 2 i wykonaj trzy następne kroki.
*Krok 2.* Zastosuj ten sam algorytm do ciągu ($l,s,x$), czyli wykonaj **MergeSort**($l,s,x$).
*Krok 3.* Zastosuj ten sam algorytm do ciagu ($s+1,p,x$), czyli wykonaj **MergeSort**($s+1,p,x$)
*Krok 4.* Zastosuj algorytm scalania **Scal** do ciągów $(x_l, ..., x_s)$, $(x_{s+1}, ..., x_p)$ i wynik umieść z powrotem w ciągu $(x_l, ..., x_p)$.

Złożoność sortowania ciągu $n$ liczb przez scalanie wynosi około $n\log n$, jest zatem znacznie mniejsza niż złożoność algorytmu sortowania przez wybór .

### 4.6 OBLICZANIE WARTOŚCI WIELOMIANU – SCHEMAT HORNERA

Obliczanie wartości wielomianu o zadanych współczynnikach jest jedną z najczęściej wykonywanych operacji w komputerze. Wynika to z ważnego faktu matematycznego, zgodnie z którym każdą funkcję (np. funkcje trygonometryczne) można zastąpić wielomianem, którego postać zależy od funkcji i od tego, jaką chcemy uzyskać dokładność obliczeń. Najefektywniejszym sposobem obliczania wartości wielomianu jest **schemat Hornera**, wynikający z następującego przekształcenia postaci wielomianu:

$$\begin{aligned} w_n(x) &= a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n = (a_0x^{n-1} + a_1x^{n-2} + ... + a_{n-1})x + a_n \\ &= ((a_0x^{n-2} + a_1x^{n-3} + ... + a_{n-2})x + a_{n-1})x + a_n = \\ &= (...((a_0x + a_1)x + a_2)x + ... + a_{n-2})x + a_{n-1})x + a_n \end{aligned}$$

Ten ostatni wzór można zapisać w następującej postaci algorytmicznej:

$$\begin{aligned} &y := a_0 \\ &y := yz + a_i \qquad \text{dla } i = 1, 2, ..., n. \end{aligned}$$

Stąd otrzymujemy algorytm:

**Algorytm: Schemat Hornera**
*Dane:* $n$ – nieujemna liczba całkowita (stopień wielomianu);
$a_0, a_1, ..., a_n$ – $n+1$ współczynników wielomianu;
$z$ – wartość argumentu.
*Wynik:* $y = w_n(z)$ – wartość wielomianu stopnia $n$ dla wartości argumentu $x = z$.
*Krok 1.* $y := a_0$ – początkową wartością jest współczynnik przy najwyższej potędze.
*Krok 2.* Dla $i = 1, 2, ..., n$ oblicz wartość dwumianu $y := yz + a_i$

Z postaci algorytmu wynika, że obliczenie wartości wielomianu stopnia $n$ wymaga wykonania $n$ mnożeń i $n$ dodawań. Udowodniono, że jest to najszybszy sposób obliczania wartości wielomianu.

Schemat Hornera ma wiele zastosowań, m.in. do:

- obliczania dziesiętnej wartości liczby, danej w systemie o podstawie $p$ – współczynniki wielomianu są w tym przypadku cyframi $\{0, 1, ..., p-1\}$ a argumentem $p$, podstawa systemu.
- szybkiego obliczania wartości potęgi (patrz punkt 5.2).

## 5 DWA TRUDNE PROBLEMY

Wracamy tutaj do dwóch problemów przedstawionych w rozdziale 3. Jednym z nich jest sprawdzanie, czy dana liczba jest liczbą pierwszą, a drugim – szybkie podnoszenie do potęgi. Dla pierwszego z tych problemów nie mamy dobrej wiadomości, nie jest bowiem znana żadna metoda szybkiego testowania pierwszości liczb. Korzysta się z tego na przykład w szyfrze RSA, częścią kluczy w tej metodzie jest liczba będąca iloczynem dwóch dużych liczb pierwszych i brak znajomości tego rozkładu jest gwarancją bezpieczeństwa tego szyfru.

Drugim problemem jest podnoszenie do dużych potęg i tutaj mamy bardzo dobrą wiadomość – podnoszenie do nawet bardzo dużych potęg, zawierających setki cyfr, trwa ułamek sekundy.

## 5.1 BADANIE ZŁOŻONOŚCI LICZB

Dla problemu badania, czy dana liczba *n* jest liczbą pierwszą, czy złożoną, dysponujemy prostym algorytmem, który polega na dzieleniu *n* przez kolejne liczby naturalne i wystarczy dzielić tylko przez liczby nie większe niż $\sqrt{n}$, gdyż liczby *n* nie można rozłożyć na iloczyn dwóch liczb większych od $\sqrt{n}$. Algorytm ten ma bardzo prostą postać:

```
var i,n:integer;
i:=2;
while i*i <= n do begin
   if n mod i = 0 then return 1;  {n jest podzielne przez i}
   i=i+1
end;
return 0   {n jest liczba pierwszą}
```

Ten fragment programu zwraca 0, jeśli *n* jest liczbą pierwszą, i 1, gdy *n* jest liczbą złożoną. W tym drugim przypadku znamy także podzielnik liczby *n*. Ten fragment programu można rozszerzyć do programu generującego wszystkie dzielniki liczby *n*.

Liczba operacji wykonywanych przez powyższy program jest w najgorszym przypadku (gdy *n* jest liczbą pierwszą) proporcjonalna do $\sqrt{n}$, a więc jeśli $n = 10^m$, to wykonywanych jest $10^{m/2}$ operacji. Zatem są niewielkie szanse, by tym algorytmem rozłożyć na czynniki pierwsze liczbę, która ma kilkaset cyfr, lub stwierdzić, że się jej nie da rozłożyć.

Rozkładem liczby złożonej na czynniki pierwsze mogą być zainteresowani ci, którzy starają się złamać szyfr RSA. Wiadomo, że jedna z liczb tworzących kluch publiczny i prywatny jest iloczynem dwóch liczb pierwszych. Znajomość tych dwóch czynników umożliwia utworzenie klucza prywatnego (tajnego). Jednak ich wielkość – są to liczby pierwsze o kilkuset cyfrach (200-300) stoi na przeszkodzie w rozkładzie *n*.

Istnieje wiele algorytmów, które służą do testowania pierwszości liczb oraz do rozkładania liczb na czynniki pierwsze. Niektóre z tych algorytmów mają charakter probabilistyczny – wynik ich działania jest poprawny z prawdopodobieństwem bliskim 1. Żaden jednak algorytm, przy obecnej mocy komputerów, nie umożliwia rozkładania na czynniki pierwsze liczb, które maja kilkaset cyfr. Szyfr RSA pozostaje więc nadal bezpiecznym sposobem utajniania wiadomości, w tym również przesyłanych w sieciach komputerowych.

## 5.2 SZYBKIE PODNOSZENIE DO POTĘGI

Wracamy tutaj do obliczania wartości potęgi $x^m$, dla dużych wartości wykładnika *m*. W punkcie 3.3 podaliśmy przykład dla $m = 12345678912345678912345678912345$. Stosując „szkolny" algorytm, czyli kolejne mnożenia, i korzystając z naszego superkomputera, obliczenie potęgi dla tego wykładnika zajęłoby $3*10^8$ lat. Ten wykładnik jest faktycznie niewielką liczbą, w porównaniu z wykładnikami, jakie pojawiają się przy stosowaniu szyfru RSA, potrzebny jest więc znacznie efektywniejszy algorytm.

Do wyprowadzenia algorytmu szybkiego potęgowania posłużymy się rekurencyjnym zapisem operacji potęgowania:

$$x^m = \begin{cases} 1 & \text{jeśli } m = 0 \\ (x^{m/2})^2, & \text{jeśli } m \text{ jest liczbą parzystą} \\ (x^{(m-1)/2})^2 x & \text{jeśli } m \text{ jest liczbą nieparzystą} \end{cases}$$

Na przykład, jeśli chcemy obliczyć $x^{22}$, to powyższa zależność rekurencyjna prowadzi przez następujące odwołania rekurencyjne: $x^{22} = (x^{11})^2 = ((x^5)^2 x)^2 = (((x^2)^2 x)^2 x)^2$.

Warto zauważyć, jaki jest związek kolejności wykonywania mnożeń, wynikającej z powyższej zależności rekurencyjnej, z postacią binarną wykładnika, zapisaną z użyciem schematu Hornera. Dla naszego przykładu mamy $m = (22)_{10} = (10110)_2$. Porównując kolejność mnożeń z postacią binarną widać, że podnoszenie do kwadratu odpowiada kolejnym pozycjom w rozwinięciu binarnym, a mnożenie przez *x* odpowiada cyfrze 1 w rozwinięciu binarnym. A zatem, liczba mnożeń jest nie większa niż dwa razy długość binarnego rozwinięcia wykładnika. Wiemy skądinąd, że długość rozwinięcia binarnego liczby *m* wynosi ok. $\log_2 m$. Mamy w przybliżeniu $\log_2 12345678912345678912345678912345 = 104$, a zatem podniesienie do tej potęgi wymaga wykonania nie więcej niż ok. 200 mnożeń.

W tabeli 3 zamieściliśmy wartości logarytmu przy podstawie 2 z rosnących wartości liczby *m*.

*Tabela 3.*
Przybliżona długość rozwinięcia binarnego liczby *m*

| $m$ | $\log_2 m$ |
|---|---|
| $10^4$ | 10 |
| $10^6$ | 20 |
| $10^9$ | 30 |
| $10^{12}$ | 40 |
| $10^{20}$ | 66,5 |
| $10^{50}$ | 166 |
| $10^{100}$ | 332,2 |

Z tabeli 3 wynika, że obliczenie wartości potęgi dla wykładnika o stu cyfrach wymaga wykonania nie więcej niż 670 mnożeń, a to trwa nawet na komputerze osobistym ułamek sekundy.

Problem potęgowania jest najlepszą ilustracją powiedzenia Ralfa Gomory'ego, że prawdziwe przyspieszenie obliczeń osiągamy dzięki efektywnym algorytmom, a nie szybszym komputerom.

## LITERATURA


1. Cormen T.H., Leiserson C.E., Rivest R.L., *Wprowadzenie do algorytmów*, WNT, Warszawa 1997
2. Gurbiel E., Hard-Olejniczak G., Kołczyk E., Krupicka H., Sysło M.M., *Informatyka, Część 1 i 2, Podręcznik dla LO*, WSiP, Warszawa 2002-2003
3. Harel D., *Algorytmika. Rzecz o istocie informatyki*, WNT, Warszawa 1992
4. Knuth D.E., *Sztuka programowania*, Tomy 1 – 3, WNT, Warszawa 2003
5. Steinhaus H., *Kalejdoskop matematyczny*, WSiP, Warszawa 1989
6. Sysło M.M., *Algorytmy*, WSiP, Warszawa 1997
7. Sysło M.M., *Piramidy, szyszki i inne konstrukcje algorytmiczne*, WSiP, Warszawa 1998. Kolejne rozdziały tej książki są zamieszczone na stronie: http://www.wsipnet.pl/kluby/informatyka_ekstra.php?k=69
8. Wirth N., *Algorytmy + struktury danych = programy*, WNT, Warszawa 1980.

W projekcie **Informatyka +**, poza wykładami i warsztatami, przewidziano następujące działania:

- 24-godzinne kursy dla uczniów w ramach modułów tematycznych
- 24-godzinne kursy metodyczne dla nauczycieli, przygotowujące do pracy z uczniem zdolnym
- nagrania 60 wykładów informatycznych, prowadzonych przez wybitnych specjalistów i nauczycieli akademickich
- konkursy dla uczniów, trzy w ciągu roku
- udział uczniów w pracach kół naukowych
- udział uczniów w konferencjach naukowych
- obozy wypoczynkowo-naukowe.

Szczegółowe informacje znajdują się na stronie projektu

**www.informatykaplus.edu.pl**