Algorytmika i programowanie

Porządek wśród informacji kluczem do szybkiego wyszukiwania

Czy wszystko można policzyć na komputerze

Dlaczego możemy się czuć bezpieczni w sieci, czyli o szyfrowaniu informacji

Znajdowanie najkrótszych dróg oraz najniższych i najkrótszych drzew

O relacjach i algorytmach

Porządek wśród informacji kluczem do szybkiego wyszukiwania

Maciej M. Sysło

Uniwersytet Wrocławski, UMK w Toruniu

syslo@ii.uni.wroc.pl, syslo@mat.uni.torun.pl

http://mmsyslo.pl/

Streszczenie

Wykład ten jest wprowadzeniem do algorytmiki i zawiera elementy implementacji algorytmów w języku programowania. Na przykładach bardzo prostych problemów przedstawione jest podejście do rozwiązywania problemów w postaci algorytmów i do ich komputerowej implementacji w języku programowania. Omawiane są m.in.: specyfikacja problemu, schematy blokowe algorytmów, podstawowe struktury danych (ciąg i tablica) oraz pracochłonność algorytmów. Wykorzystywane jest oprogramowanie edukacyjne, ułatwiające zrozumienie działania algorytmów i umożliwiające wykonywanie eksperymentów z algorytmami bez konieczności ich programowania. Przytoczono ciekawe przykłady zastosowań omawianych zagadnień.

Zakres tematyczny tego wykładu obejmuje problemy poszukiwania elementów (informacji) w zbiorach nieuporządkowanych i uporządkowanych oraz problem porządkowania (sortowania), któremu poświęcamy szczególnie wiele miejsca ze względu na jego znaczenie w obliczeniach. Algorytmy sortowania są okazją, by na ich przykładzie zademonstrować różne techniki rozwiązywania problemów, jak metodę dziel i zwyciężaj oraz rekurencję.

Spis treści

Część materiałów pochodzi z książek, których Maciej M. Sysło jest autorem lub współautorem [2, 7, 8]. Oprogramowanie edukacyjne wykorzystane w materiałach można pobrać ze strony http://mmsysło.pl/. Tabele i rysunki pochodzą z książek autora lub są przez niego opracowane.

1 PRZESZUKIWANIE ZBIORU

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?
- poszukaj w swojej klasie ucznia, któremu droga do szkoły zabiera najwięcej czasu;
- znajdź najstarszego ucznia w swojej szkole; jak zmieni się Twój algorytm, gdybyś chciał znaleźć w szkole najmłodszego ucznia?
- odszukaj największą kartę w potasowanej talii kart;
- znajdź najlepszego gracza w warcaby w swojej klasie (zakładamy, że wszyscy potrafią grać w warcaby);
- odnajdź najlepszego tenisistę w swojej klasie.

Tego typu problemy pojawiają się bardzo często, również w obliczeniach komputerowych, i na ogół są rozwiązywane w dość naturalny sposób – przeglądany jest cały zbiór, by znaleźć poszukiwany element. Zastanowimy się, jak dobra jest to metoda, i czy może istnieje szybsza metoda znajdowania w zbiorze elementu o określonych własnościach.

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. To 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. Metodzie pucharowej odpowiada **drzewo algorytmu**, służące wyjaśnieniu wielu innych kwestii związanych głównie ze złożonością algorytmów.

Założenia

Poczyńmy najpierw pewne założenia.

Założenie 1. Na początku wykluczamy, że przeszukiwane zbiory elementów są uporządkowane, np. klasa – od najwyższego do najniższego ucznia lub odwrotnie, szkoła – od najmłodszego do najstarszego ucznia lub odwrotnie, Gdyby tak było, to rozwiązanie wymienionych wyżej problemów i im podobnych byłoby dziecinnie łatwe – wystarczyłoby wziąć element z początku albo z końca takiego uporządkowania.

Założenie 2. 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. Przeszukiwaniem zbiorów uporządkowanych zajmiemy się w punkcie 6.2.

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. Co więcej, w większości wymienionych problemów porównanie elementów sprowadza się do porównania liczb, właściwych dla branych pod uwagę elementów, a oznaczających: wzrost, czas na dojście do szkoły (np. liczony w minutach), wiek. Elementy zbiorów utożsamiamy więc z ich wartościami i wartości te nazywamy **danymi**, chociaż często prowadzimy rozważania w języku problemu, posługując się nazwami elementów: wzrost, wiek itp.

Przy porównywaniu kart należy uwzględnić ich kolory i wartości. Natomiast, by znaleźć w klasie najlepszego gracza w tenisa, należy zorganizować turniej – ten problem omówimy w dalszej części wykładu.

1.1 SPECYFIKACJA PROBLEMU I ALGORYTM

Dla uproszczenia rozważań można więc założyć, że dany jest pewien zbiór liczb *A* i w tym zbiorze należy znaleźć liczbę najmniejszą (lub największą). Przyjmijmy, że tych liczb jest *n* i oznaczmy je jako ciąg liczb: $x_1, x_2, \ldots, x_n$. Możemy teraz podać **specyfikację** 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

Dla powyższej specyfikacji podamy teraz algorytm, który polega na przejrzeniu ciągu danych od początku do końca. Opis algorytmu poprzedza specyfikację problemu, który ten algorytm rozwiązuje – tak będziemy na ogół postępować w każdym przypadku. Przedstawiony poniżej opis algorytmu ma postać **listy kroków**. O innych sposobach przedstawiania algorytmów piszemy w dalszej części.

Algorytm 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*.

Krok 1. Przyjmij za *min* pierwszy element w zbiorze (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 $min := x_i$.

Uwaga. W opisie algorytmu pojawiło się polecenie (instrukcja) **przypisania**[13], np. $min := x_1$, w której występuje symbol :=, złożony z dwóch znaków: dwukropka i równości. Przypisanie oznacza nadanie wielkości zmiennej stojącej po lewej stronie tego symbolu wartości równej wartości wyrażenia (w szczególnym przypadku to wyrażenie może być zmienną) znajdującego się po prawej stronie tego symbolu. Przypisanie jest stosowane na przykład wtedy, gdy należy zmienić wartość zmiennej, np. $i := i + 1$ – w tym przypadku ta sama zmienna występuje po lewej i po prawej stronie symbolu przypisania. Polecenie przypisania występuje w większości języków programowania, stosowane są tylko różne symbole i ich uproszczenia dla jego oznaczenia. W schemacie blokowym na rysunku 2 symbolem przypisania jest strzałka ←.

Metoda zastosowana w algorytmie **Min**, polegająca na badaniu elementów ciągu danych w kolejności, w jakiej są ustawione, nazywa się **przeszukiwaniem liniowym**, w odróżnieniu od przeszukiwania przez połowienie (lub binarnego), o którym jest mowa w rozdziale 6.

Demonstracja działania algorytmu Min

Działanie algorytmu **Min** można zademonstrować posługując się programem edukacyjnym **Maszyna sortująca** (patrz rys. 1), który jest udostępniony wraz z tymi materiałami. W tym programie można ustalić liczbę elementów w ciągu (między 1 i 16) i wybrać rodzaj ciągu danych, który może zawierać elementy: losowe, posortowane rosnąco lub posortowane malejąco. Ponieważ ten program służy do porządkowania ciągu, o czym będzie mowa w dalszej części zajęć (patrz rozdz. 4), zalecamy tutaj wykonanie demonstracji pracą krokową i przerwanie jej po znalezieniu najmniejszego elementu w ciągu – następuje to w momencie, gdy dolna zielona strzałka znajdzie się pod ostatnim elementem w ciągu i wygaszony zostanie czerwony kolor, wyróżniający porównywane elementy.

[13] Polecenie przypisania jest czasem nazywane niepoprawnie podstawieniem.

Algorytm Max – Prosta modyfikacja algorytmu Min

Podany powyżej algorytm **Min**, służący do znajdowania najmniejszej liczby w ciągu danych, może być łatwo zmodyfikowany do znajdowania największego elementu ciągu – wystarczy w tym celu zmienić tylko zwrot nierówności.

Jeszcze jedna modyfikacja algorytmu Min

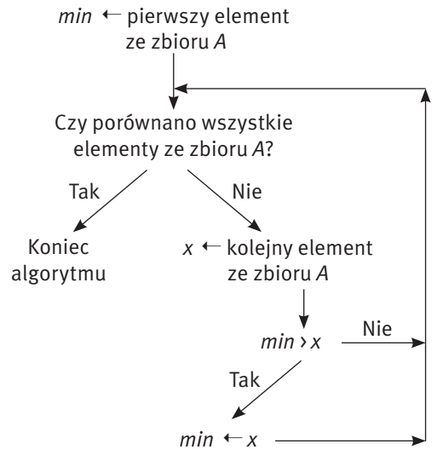
Często, poza znalezieniem elementu najmniejszego (lub największego), chcielibyśmy znać jego położnie, czyli miejsce (numer) w ciągu danych. W tym celu wystarczy wprowadzić nową zmienną, np. *imin*, w której będzie przechowywany numer aktualnie najmniejszego elementu – szczegóły tej modyfikacji pozostawiamy do samodzielnego wykonania.

Rysunek 1.
Demonstracja działania algorytmu **Min** w programie **Maszyna sortująca**

1.2 SCHEMAT BLOKOWY ALGORYTMU MIN

Schemat blokowy algorytmu (zwany również **siecią działań** lub **siecią obliczeń**) jest graficznym opisem: działań składających się na algorytm, ich wzajemnych powiązań i kolejności ich wykonywania. W informatyce miejsce schematów blokowych jest pomiędzy opisem algorytmu w postaci listy kroków, a programem, napisanym w wybranym języku programowania. Należą one do kanonu wiedzy informatycznej, nie są jednak niezbędnym jej elementem, chociaż mogą okazać się bardzo przydatne na początkowym etapie projektowania algorytmów i programów komputerowych. Z drugiej strony, w wielu publikacjach algorytmy są przedstawiane w postaci schematów blokowych, pożądana jest więc umiejętność ich odczytywania i rozumienia. Ten sposób reprezentowania algorytmów pojawia się również w zadaniach maturalnych z informatyki.

Na rysunku 2 przedstawiono schemat algorytmu **Min**. Jest to bardziej schemat ideowy działania algorytmu niż jego schemat blokowy, jest bardzo ogólny, gdyż zawarto w nim jedynie najważniejsze polecenia i pominięto szczegóły realizacji poszczególnych poleceń.



Rysunek 2.
Pierwszy, zgrubny schemat blokowy algorytmu **Min**

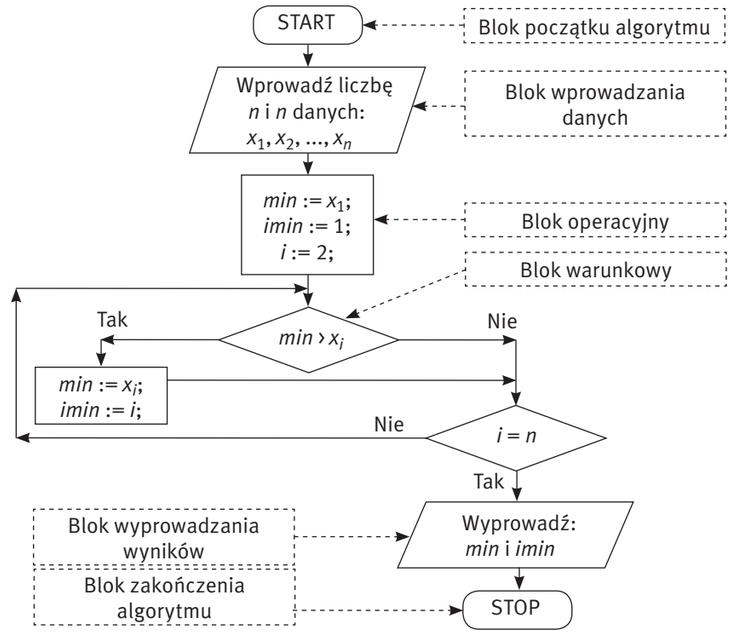
Na rysunku 3 przedstawiony jest szczegółowy schemat blokowy algorytmu **Min**, uwzględniono w nim również modyfikację zaproponowaną pod koniec poprzedniego punktu oraz zawarto bloki wczytywania danych i wyprowadzania wyników. Jest on zbudowany z bloków, których kształty zależą od rodzaju wykonywanych w nich poleceń. I tak mamy:

- blok początku i blok końca algorytmu;
- blok wprowadzania (wczytywania) danych i wyprowadzania (drukowania) wyników – bloki te mają taki sam kształt;
- blok operacyjny, w którym są wykonywane operacje przypisania;
- blok warunkowy, w którym jest formułowany warunek;
- blok informacyjny, który może służyć do komentowania fragmentów schematu lub łączenia ze sobą części większych schematów blokowych.

Nie istnieje pełny układ zasad poprawnego konstruowania schematów blokowych. Można natomiast wymienić dość naturalne zasady, wynikające z charakteru bloków:

- schemat zawiera dokładnie jeden blok początkowy, ale może zawierać wiele bloków końcowych – początek algorytmu jest jednoznacznie określony, ale algorytm może się kończyć na wiele różnych sposobów;
- z bloków: początkowego, wprowadzania danych, wyprowadzania wyników, operacyjnego wychodzi dokładnie jedno połączenie, może jednak wchodzić do nich wiele połączeń;
- z bloku warunkowego wychodzą dwa połączenia, oznaczone wartością warunku: TAK i NIE;
- połączenia wychodzące mogą dochodzić do bloków lub do innych połączeń.

Schematy blokowe mają wady trudne do wyeliminowania. Łatwo konstruuje się z ich pomocą algorytmy dla obliczeń niezawierających iteracji i warunków, którym w schematach odpowiadają rozgałęzienia, nieco trudniej dla obliczeń z rozgałęzieniami, a trudniej dla obliczeń iteracyjnych (wczytywanie ciągu i realizacja kroku 2 z algorytmu **Min**). Za pomocą schematów blokowych nie można w naturalny sposób zapisać rekurencji oraz objaśnić znaczenia wielu pojęć związanych z algorytmiką, takich np. jak programowanie z użyciem procedur, czyli podprogramów z parametrami.



Rysunek 3.
Szczegółowy schemat blokowy algorytmu **Min**

1.3 KOMPUTEROWA REALIZACJA ALGORYTMU

Reprezentowanie danych w algorytmach

Zanim podamy komputerową realizację pierwszego algorytmu, musimy ustalić, w jaki sposób będą reprezentowane w algorytmie dane i jak będziemy je podawać do algorytmu.

Wspomnieliśmy już przy projektowaniu algorytmu **Min**, że dane dla tego algorytmu są zapisane w postaci ciągu n liczb $x_1, x_2, \ldots, x_n$. Liczby te mogą być naturalne (czyli całkowite i dodatnie), całkowite lub dziesiętne (np. z kropką). Rodzaj danych liczb nazywa się **typem danych**. Przyjmujemy dla uproszczenia, że danymi dla algorytmów omawianych na tych zajęciach są liczby całkowite.

Zbiór danych, który jest przetwarzany za pomocą algorytmu, może być podawany (czytany) z klawiatury, czytany z pliku lub może być zapisany w innej strukturze danych.

Dla wygody będziemy zakładać, że wiemy, ile będzie danych i ta liczba danych występuje na początku danych – jest nią liczba n w opisie algorytmu **Min**. W ogólności, jeśli np. dane napływają do komputera z jakiegoś urządzenia pomiarowego, możemy nie wiedzieć, ile ich będzie. W takich przypadkach wprowadza się dane aż do specjalnego znaku, który świadczy o ich końcu. Takim znakiem może być koniec pliku, jeśli dane są umieszczone w pliku. Może nim być również wyróżniona liczba zwana **wartownikiem**, której rolą jest pilnowanie końca danych. O użyciu wartownika powiemy później (punkt 6.1), a teraz przedstawimy realizację algorytmu **Min** dla danych podawanych z klawiatury.

Uwaga. Piszemy „zbiór danych”, ale użycie tutaj pojęcia zbiór nie zawsze jest matematycznie poprawne. W zbiorze elementy się nie powtarzają, a w danych mogą występować takie same liczby. Całkowicie poprawnie powinniśmy mówić o tzw. **multizbiorach**, czyli zbiorach, w których elementy mogą się powtarzać, ale dla wygody będziemy stosować pojęcie zbioru, pamiętając, że mogą powtarzać się w nim elementy. Sytuację

upraszcza nam założenie, że zbiór danych w algorytmie będzie przedstawiony w postaci ciągu elementów, a w ciągu elementy mogą się powtarzać.

Komputerowa realizacja algorytmu Min – dane z klawiatury

Zapiszemy teraz algorytm **Min** posługując się poleceniami języka Pascal. Przyjmujemy, że dane są podawane z klawiatury – na początku należy wpisać liczbę wszystkich danych, a po niej kolejne elementy danych w podanej ilości. Po każdej danej liczbie naciskamy klawisz Enter. Program, który jest zapisem algorytmu **Min** w języku Pascal, jest umieszczony w drugiej kolumnie w tabeli 1. Język Pascal jest zrozumiały dla komputerów, które są wyposażone w specjalne programy, tzw. **kompilatory**, przekładające programy użytkowników na język wewnętrzny komputerów. Program w tabeli 1 bez większego trudu zrozumie także człowiek dysponujący opisem algorytmu **Min** w postaci listy kroków. W wierszu nr 2 znajdują się **deklaracje**, czyli opisy zmiennych – komputer musi wiedzieć, jakimi wielkościami posługuje się algorytm i jakiego są one **typu**, `integer` oznacza liczby całkowite. Polecenia w językach programowania nazywają się **instrukcjami**. Jeśli chcemy z kilku instrukcji zrobić jedną, to tworzymy z nich **blok**, umieszczając na jego początku słowo `begin`, a na końcu – `end`. Pojedyncze instrukcje kończymy **średnikiem**. Na końcu programu stawiamy kropkę.

Ada Augusta, córka Byrona, uznawana powszechnie za pierwszą programistkę komputerów, przełomowe znaczenie maszyny analitycznej Ch. Babbage'a, pierwowzoru dzisiejszych komputerów, upatrywała właśnie „w możliwości wielokrotnego wykonywania przez nią danego ciągu instrukcji, z liczbą powtórzeń z góry zadaną lub zależną od wyników obliczeń", a więc w iteracji.

Dwie instrukcje wymagają wytłumaczenia, chociaż również są dość oczywiste. W wierszach 6–11 znajdują się instrukcje, które realizują krok 2 algorytmu, polegający na wykonaniu wielokrotnie sprawdzenia warunku. Instrukcja, służąca do wielokrotnego wykonania innych instrukcji nazywa się **instrukcją iteracyjną** lub **instrukcją pętli**. W programie w tabeli 1 ta instrukcja zaczyna się w wierszu nr 6, a kończy w wierszu nr 11:

```
for i:=2 to n do begin
...
end
```

Ta instrukcja iteracyjna służy do powtórzenia **instrukcji warunkowej**, która zaczyna się w wierszu nr 8 i kończy w wierszu nr 10. Ma tutaj postać:

```
if min>x then begin
...
end
```

Inne typy instrukcji iteracyjnej i warunkowej będą wprowadzane sukcesywnie.

1.4. PRACOCHŁONNOŚĆ (ZŁOŻONOŚĆ) ALGORYTMU MIN

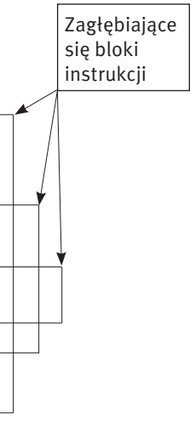
Problem znajdowania najmniejszego (lub największego) elementu w zbiorze jest jednym z elementarnych problemów najczęściej rozwiązywanych przez człowieka i przez komputer, dlatego interesujące jest pytanie, czy rozwiązujemy go możliwie najszybciej. W szczególności, czy podany przez nas algorytm **Min** i jego komputerowe **implementacje**[14] są najszybszymi metodami rozwiązywania tego problemu.

W algorytmach **Min** i **Max**, i w ich implementacjach, podstawową operacją jest porównanie dwóch elementów ze zbioru danych – policzmy więc, ile porównań jest wykonywanych w tych algorytmach. Liczba tych porównań w algorytmie zależy od liczby danych. W każdej iteracji algorytmu jest wykonywane jedno porównanie $min > x_i$, a zatem w każdym z tych algorytmów jest wykonywanych $n - 1$ porównań (tyle razy bowiem

[14] Terminem **implementacja** określa się w informatyce komputerową realizację algorytmu.

Tabela 1.
Program w języku Pascal (druga kolumna)

Lp	Program w języku Pascal	Odpowiedniki instrukcji po polsku
1.	`Program MinKlawiatura;`	nazwa programu
2.	`var i,imin,min,n,x:integer;`	deklaracja zmiennych: i, imin, min, n, x
3.	`begin`	początek głównego bloku programu
4.	`read(n);`	czytaj(n);
5.	`read(x); min:=x; imin:=1;`	czytaj(x); początek szukania min
6.	`for i:=2 to n do begin`	dla i:=2 do n wykonaj – początek iteracji
7.	`read(x);`	czytaj(x);
8.	`if min>x then begin`	jeśli min>x to – instrukcja warunkowa
9.	`min:=x; imin:=i`	min:=x; imin:=i
10.	`end`	koniec instrukcji warunkowej
11.	`end;`	koniec iteracji
12.	`write(imin,min)`	drukuj(imin, min)
13.	`end.`	koniec. – na końcu stawiamy kropkę



jest wykonywana iteracja w kroku 2). 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:

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

To nie jest specjalnie wielkie odkrycie, a jedynie sformułowanie dość oczywistej własności postępowania, które często wykonujemy niemal automatycznie, nie zastanawiając się nawet, w jaki sposób to robimy. Już większym wyzwaniem jest pytanie:

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[15], 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 Tomek 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 minimum jeden mecz, czyli rozegranych zostało przynajmniej tyle meczów, ilu jest uczniów w klasie mniej jeden. I to kończy nasze uzasadnienie.

Z dotychczasowych rozważań możemy wyciągnąć wniosek, że algorytmy **Min** i **Max** oraz ich komputerowe implementacje są najlepszymi algorytmami służącymi do znajdowania najmniejszego i największego elementu, gdyż wykonywanych jest w nich tyle porównań, ile musi wykonać jakikolwiek algorytm rozwiązywania tych problemów. O takim algorytmie mówimy, że jest **algorytmem optymalnym pod względem złożoności obliczeniowej**.

[15] Posługujemy się tutaj argumentacją zaczerpniętą z książki Hugona Steinhausa, [rozdz. III, 6].

2 KOMPLETOWANIE PODIUM ZWYCIĘZCÓW TURNIEJU

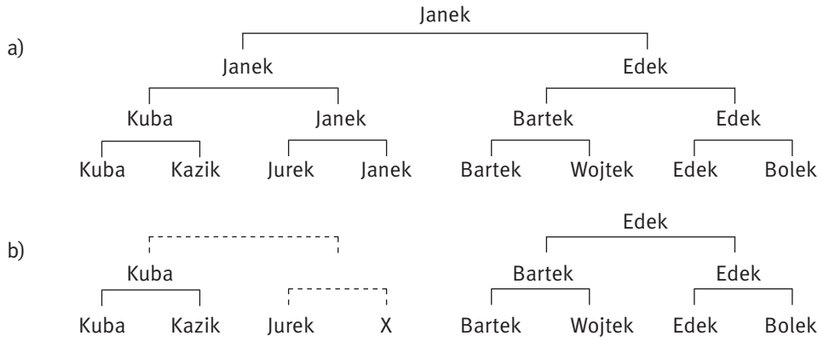
Przedstawione w poprzednim rozdziale postępowanie nie jest jedyną metodą służącą do znajdowania najlepszego elementu w zbiorze. Inną metodą jest tzw. **system pucharowy**, stosowany często przy wyłanianiu najlepszego zawodnika bądź drużyny w turnieju. W metodzie tej „porównanie" dwóch zawodników (lub drużyn), by stwierdzić, który jest lepszy („większy"), polega na rozegraniu meczu. W rozgrywkach systemem pucharowym zakłada się, że wszystkie mecze kończą się zwycięstwem jednego z zawodników, dlatego w dalszej części będziemy pisać o rozgrywkach w tenisa, a nie o turnieju w warcaby, gdyż w przypadku warcabów (jak i szachów) partie mogą kończyć się remisem, podczas gdy w meczach w tenisa można wymóc, by mecz między dwoma zawodnikami zawsze kończył się zwycięstwem jednego z nich.

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 4a) 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 mniej, niż wystąpiło w turnieju zawodników. Nie jest to przypadek. Ten fakt jest prawdziwy dla dowolnej liczby zawodników występujących w turnieju, rozgrywanym metodą pucharową – sprawdź dla 6, 7, 9, 13, 15 zawodników.

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.



Rysunek 4.
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. Zastanawiał się on bowiem, jaka jest najmniejsza liczba meczów tenisowych do rozegrania w grupie zawodników, niezbędna do tego, aby wyłonić wśród nich najlepszego i drugiego najlepszego zawodnika. Wtedy, tak jak i dzisiaj, rozgrywano turnieje tenisowe systemem pucharowym. Zapewnia on, że zwycięzca finału jest najlepszym zawodnikiem, gdyż pokonał wszystkich uczestników turnieju: niektórych bezpośrednio – wygrywając z nimi w spotkaniach,

Pamiętaj. Wicemistrz wyłoniony systemem pucharowym na ogół nie jest drugą najlepszą drużyną (zawodnikiem) turnieju.

a niektórych pośrednio – pokonując ich zwycięzców. W takich 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.

By się przekonać, że wątpliwości H. Steinhausa były rzeczywiście uzasadnione, spójrzmy na drzewo turnieju przedstawione na rysunku 4a). 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, jak Edek by 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. Hugo Steinhaus oczywiście znał to rozwiązanie, pytał więc o najmniejszą potrzebną liczbę meczów, i takiej odpowiedzi udzielił w 1932 inny polski matematyk Józef Schreier, chociaż jego dowód nie był w pełni poprawny i został skorygowany dopiero po 32 latach (w 1964 roku przez Sergeia Sergeyevicha Kislitsyna).

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 4a). 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 4b) oznaczyliśmy przerywaną 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). A zatem, jaka jest wysokość drzewa turnieju? 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.

Porównaj wartości dwóch wyrażeń: $2n-3$ oraz $n + \log_2 n - 2$, odpowiadających liczbie porównań wykonywanych w dwóch omówionych wyżej algorytmach znajdowania najlepszego i drugiego najlepszego zawodnika turnieju. Dla ułatwienia obliczeń przyjmij, że n jest potęgą liczby 2.

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

Na naszym podium zwycięzców turnieju tenisowego brakuje jeszcze trzeciego najlepszego zawodnika, czyli kogoś, kto jest lepszy od wszystkich pozostałych zawodników z wyjątkiem już wyłonionych – najlepszego i drugiego najlepszego. Znalezienie go bardzo przypomina wyłanianie drugiego najlepszego zawodnika. Przy-

puśćmy, że w naszym przykładowym turnieju, drugie miejsce zajął Kuba. W jaki sposób należy zorganizować dogrywkę, by wyłonić trzeciego najlepszego zawodnika turnieju. A jeśli drugie miejsce zajął jednak Edek – jak należy postępować w tym przypadku? Sformułuj ogólną zasadę i oblicz, ile meczów należy rozegrać, by wyłonić zawodnika zajmującego trzecie miejsce?

To postępowanie można kontynuować wyznaczając czwartego, piątego itd. zawodnika turnieju. Ostatecznie otrzymamy pełne uporządkowanie wszystkich zawodników biorących udział w turnieju. Taka metoda nazywa się **porządkowaniem na drzewie** i może być stosowana również do porządkowania liczb.

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. Im większa jest rozpiętość, tym większy jest rozrzut wartości elementów zbioru. Interesujące jest więc jednoczesne znalezienie najmniejszej i największej wartości w zbiorze liczb.

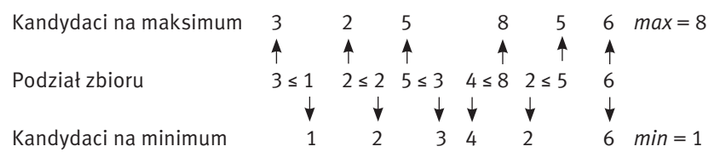
Rozwiązanie naiwne

Na podstawie dotychczasowych rozważań, dotyczących wyznaczania najmniejszej i największej wartości w zbiorze liczb, zapewne łatwo podasz algorytm znajdowania jednocześnie obu tych elementów w zbiorze. Ile należy w tym celu wykonać porównań?

Odpowiedź na to pytanie ilustruje częste podejście, stosowane w matematyce i informatyce, które polega na tym, że w rozwiązaniu nowego problemu korzystamy ze znanej już metody. Stosujemy więc 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?

Rozwiązanie metodą dziel i zwyciężaj

Postaramy się znacznie przyspieszyć to postępowanie, a będzie to polegało na rzeczywiście jednoczesnym szukaniu najmniejszego i największego elementu w całym zbiorze, jak również wykorzystaniu poznanego algorytmu znajdowania tych elementów w pewnych podzbiorach rozważanego zbioru. W tym celu rozważmy ponownie podstawową operację – porównanie elementów – i 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 odwrotna, 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. Pewnym problemem jest to, co zrobić z ostatnim elementem ciągu, gdy zbiór ma nieparzystą liczbę elementów. W tym przypadku możemy dodać ten element do jednego i do drugiego podzbioru kandydatów. Postępowanie to jest zilustrowane przykładem na rysunku 5.



Rysunek 5.
Przykład postępowania podczas jednoczesnego znajdowania minimum i maksimum w ciągu liczb

Zapiszmy opisane postępowanie w postaci algorytmu poprzedzając go specyfikacją.

Algorytm Min-i-Max – jednoczesne znajdowanie największego i 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 liczba *min* i największa liczba *max* wśród liczb $x_1, x_2, \ldots, x_n$.
Krok 1. {Podział zbioru danych na dwa podzbiory: M – zbiór kandydatów na minimum i N – zbiór kandydatów na maksimum. Na początku te zbiory są puste.}
Jeśli n jest liczbą parzystą, to dla $i = 1, 3, \ldots, n - 1$, a jeśli n jest liczbą nieparzystą, to dla $i = 1, 3, \ldots, n - 2$ wykonaj:
jeśli $x_i \le x_{i+1}$, to dołącz x_i do M, a x_{i+1} do N,
a w przeciwnym razie dołącz x_i do N, a x_{i+1} do M.
Jeśli n jest liczbą nieparzystą, to dołącz x_n do obu zbiorów M i N.
Krok 2. Znajdź *min* w zbiorze M, stosując algorytm **Min**.
Krok 3. Znajdź *max* w zbiorze N, stosując algorytm **Max**.

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

- podziału danych na dwa podzbiory równoliczne (krok 1);
- zastosowania znanych już algorytmów **Min** i **Max** do utworzonych podzbiorów danych (kroki 2 i 3).

Jest to przykład zasady (metody) rozwiązywania problemów. która wielokrotnie pojawia się na zajęciach z algorytmiki i programowania. Nosi ona nazwę **dziel i zwyciężaj** i jest jedną z najefektywniejszych metod algorytmicznych w informatyce (patrz rozdz. 7). **Dziel** – odnosi się do podziału zbioru danych na podzbiory, zwykle o jednakowej liczbie elementów, do których następnie są stosowane odpowiednie algorytmy. **Zwyciężstwo** – to efekt końcowy, czyli efektywne rozwiązanie rozważanego problemu.

Pracochłonność jednoczesnego znajdowania minimum i maksimum

Obliczmy, ile porównań między elementami danych jest wykonywanych w algorytmie **Min-i-Max**. Rozważmy najpierw przypadek, gdy n jest liczbą parzystą. W takim przypadku, 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**[16], czyli najmniejszą liczbę całkowitą k spełniającą nierówność $x \le k$.

A zatem, w algorytmie **Min-i-Max** wykonuje się $\lceil 3n/2 \rceil - 2$ porównania, czyli ok. $n/2$ mniej porównań niż w algorytmie naiwnym, podanym na początku tego punktu.

Zastosowana tutaj zasada **dziel i zwyciężaj** jest na ogół stosowana w sposób **rekurencyjny** – problem jest dzielony na podproblemy, te są ponownie dzielone na podproblemy, i tak dalej, aż do otrzymania podproblemów, dla których rozwiązanie można łatwo wskazać, np. gdy liczba danych w podproblemie wynosi 1 lub 2. Tę ogólną metodę dziel i zwyciężaj ilustrujemy dalej na przykładach poszukiwania elementu w zbiorze uporządkowanym (punkt 6.2) oraz sortowania przez scalanie.

4 PROBLEM PORZĄDKOWANIA – PORZĄDKOWANIE PRZEZ WYBÓR

Porządkowanie, nazywane również często **sortowaniem** (będziemy tych terminów używali zamiennie) ma olbrzymie znaczenie niemal w każdej działalności człowieka. Jeśli elementy w zbiorze są uporządkowane

[16] Funkcja **powała** (i towarzysząca jej funkcja **podłoga**) odgrywają ważną rolę w rozważaniach informatycznych.

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:

- 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 – ludzi, gdy posługujemy się nimi, 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 (np. w programie Eksplorator Windows) i wtedy na ogół znajdujemy odpowiedź natychmiast. Porządkowanie jest również podstawową operacją wykonywaną na dużych zbiorach informacji, np. w bazach danych.

Często porządkujemy różne elementy lub wykonujemy powyższe operacje na uporządkowanych zbiorach nie korzystając z komputera – w tym również mogą nam pomóc metody porządkowania i algorytmy działające na uporządkowanych zbiorach omówione na zajęciach komputerowych.

4.1 PROBLEM PORZĄDKOWANIA

Na tych zajęciach będziemy zajmować się głównie porządkowaniem liczb, chociaż wiele praktycznych problemów dotyczy porządkowania innych obiektów przechowywanych w komputerze. Przyjmijmy więc następującą specyfikację tego problemu.

Problem porządkowania (sortowania)
Dane: Liczba naturalna n i ciąg n liczb $x_1, x_2, \ldots, x_n$.
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.

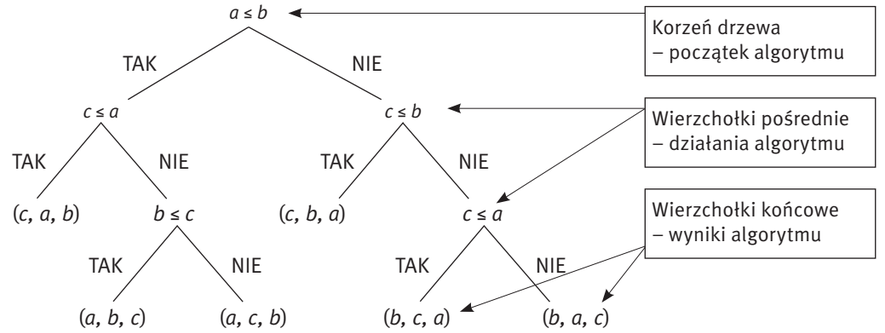
W ogólnym sformułowaniu problemu porządkowania nie czynimy żadnych założeń dotyczących elementów, które porządkujemy, ważna jest jedynie relacja między wartościami liczb. W ogólnym przypadku na ogół zakładamy, że porządkowane liczby są całkowite dodatnie. W wielu sytuacjach praktycznych porządkowane są najróżniejsze obiekty, np. słowa (przy tworzeniu słownika), adresy (do książki telefonicznej) czy bardzo złożone zestawy danych, jak np. informacje o koncie bankowym i jego właścicielu. Szczególnym przypadkiem porządkowania liczb jest sytuacja, w której liczby mają niewielkie wartości – specjalny algorytm porządkowania takich liczb jest zamieszczony w rozdziale 5.

4.2 PORZĄDKOWANIE KILKU ELEMENTÓW

Jeśli liczba elementów w ciągu jest mała, np. $n = 2, 3, 4$, to łatwo można podać algorytmy, w których jest wykonywana możliwie najmniejsza liczba porównań.

Mając dwa elementy, wystarczy jedno porównanie, by ustawić je w porządku. Nieco więcej zachodu wymaga porządkowanie trzech liczb. Mając do uporządkowania trzy liczby, możemy postąpić następująco: najpierw ustawić a i b w odpowiedniej kolejności, a potem wstawić c w odpowiednie miejsce względem a i b. W pierwszym etapie wykonujemy jedno porównanie, a w drugim? Załóżmy, że po pierwszym porównaniu, czyli $a \le b$, otrzymujemy uporządkowanie (a,b). Wtedy, jeśli $c \le a$, to (c,a,b) jest szukanym uporządkowaniem. W przeciwnym razie musimy jeszcze sprawdzić, czy $c \le b$. Jeśli tak, to otrzymujemy uporządkowanie (a,c,b), a w przeciwnym razie – (a,b,c). Podobnie postępujemy, gdy po pierwszym porównaniu kolejność liczb a i b jest (b,a).

Wszystkie te przypadki można zapisać w postaci **drzewa algorytmu**, przedstawionego na rysunku 6. Drzewo algorytmu jest pewnego rodzaju „schematem blokowym" algorytmu – w sposób graficzny ilustruje przebieg działania algorytmu dla dowolnych danych. Ta reprezentacja algorytmu jest bardzo przydatna do śledzenia, jak algorytm działa dla poszczególnych danych. Wygodna jest również do analizy pracochłonności algorytmu, czyli do wyznaczania, ile porównań wykonuje algorytm dla poszczególnych danych.



Rysunek 6.
Drzewo algorytmu porządkującego trzy liczby

Wykonywanie algorytmu, zapisanego w postaci drzewa, rozpoczyna się w **korzeniu** tego drzewa, przechodzi przez **wierzchołki pośrednie**, odpowiadające operacjom wykonywanym w algorytmie, i kończy się w **wierzchołku końcowym**. Wierzchołki końcowe drzewa algorytmu zawierają wszystkie możliwe rozwiązania – w naszym przykładzie są to wszystkie możliwe uporządkowania trzech elementów. Takich uporządkowań jest 6.

Może się zdarzyć, że porządkowane liczby są uporządkowane, mogą się również powtarzać – algorytm porządkowania powinien dawać poprawną odpowiedź również w tych przypadkach, czyli działać jak „bezmyślny automat" do wykonywania obliczeń. W tym konkretnym przypadku nie powinien „zauważyć", że liczby są już uporządkowane i przystąpić do działania, jak dla dowolnej innej trójki liczb. Jest to potwierdzenie **uniwersalności** algorytmu porządkowania trzech liczb – może on być stosowany do dowolnej trójki liczb, żadnej specjalnie nie wyróżniając.

Algorytm porządkowania trzech liczb może być łatwo rozszerzony do algorytmu porządkowania czterech liczb przez wstawienie czwartej liczby do już uporządkowanego ciągu trzech liczb w każdym wierzchołku wiszącym drzewa – wymaga to wykonania dodatkowo dwóch porównań.

Niestety, w podobny sposób nie da się otrzymać najlepszego algorytmu porządkowania pięciu liczb – odsyłamy do książki [7], gdzie szczegółowo opisano taki algorytm, który wykonuje 7 porównań w najgorszym przypadku.

Wspomnieliśmy, że drzewo algorytmu ułatwia analizę pracochłonności algorytmu. Na podstawie drzewa z rysunku 6 wynika, że przy porządkowaniu trzech liczb trzeba wykonać co najwyżej 3 porównania – jest to bowiem najdłuższa (w sensie liczby wierzchołków pośrednich, które odpowiadają działaniom w algorytmie) droga z korzenia do wierzchołka końcowego. Długość takiej drogi nazywamy **wysokością drzewa**.

4.3 PORZĄDKOWANIE PRZEZ WYBÓR

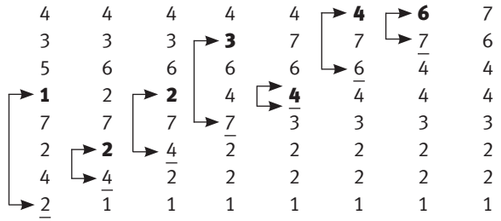
Zajmiemy się teraz porządkowaniem ciągów, które mogą zawierać dowolną liczbę elementów. Wykorzystamy w tym celu jeden z poznanych wcześniej algorytmów. O innych algorytmach porządkowania wspominamy w dalszej części wykładu.

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**.

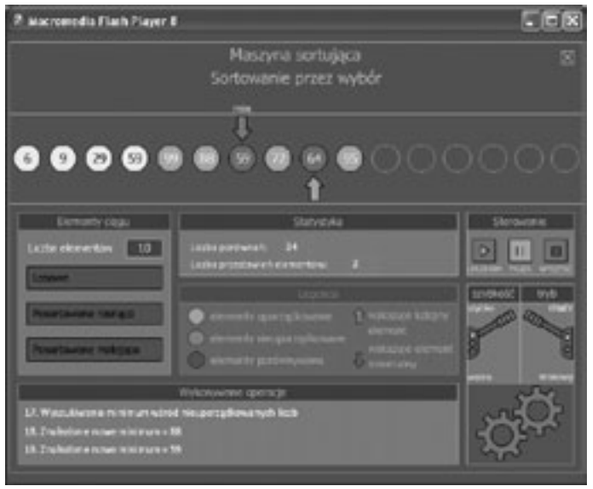
Demonstracja działania porządkowania przez wybór

Aby zilustrować działanie tego algorytmu, załóżmy, że ciąg elementów, który mamy uporządkować, jest zapisany w kolumnie (patrz rys. 7). Chcemy ponadto, aby wynik, czyli ciąg uporządkowany, znalazł się w tym samym ciągu – o takim algorytmie mówimy, że działa ***in situ***, czyli „w miejscu". W tym celu wystarczy znaleziony najmniejszy element w ciągu zamienić miejscami z pierwszym elementem tego ciągu. Rysunek 7 ilustruje kolejne kroki działania algorytmu porządkowania przez wybór.



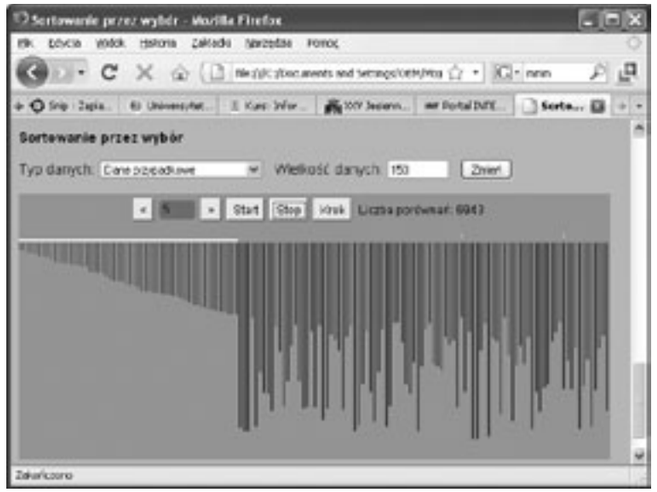
Rysunek 7.
Ilustracja działania algorytmu porządkowania przez wybór. W każdej kolumnie, pogrubiony został najmniejszy element w podciągu od góry do kreski, a klamra wskazuje zamianę elementów miejscami

Zanim podamy szczegółowy opis tego algorytmu porządkowania, przyjrzyj się pełnej demonstracji jego działania w programie **Maszyna sortująca**, który był wykorzystany do demonstracji działania algorytmu znajdującego najmniejszy element w ciągu.



Rysunek 8.
Demonstracja działania algorytmu porządkowania przez wybór w programie **Maszyna sortująca** – cztery pierwsze elementy znajdują się już na swoim miejscu w ciągu uporządkowanym i szukany jest najmniejszy element w pozostałej części ciągu, by przenieść go na miejsce piąte

Polecamy również inny program **Sortowanie**, który służy do demonstracji działania oraz porównywania między sobą wielu algorytmów porządkujących. Ten program jest również załączony do materiałów do tych zajęć i może być wykorzystany w celach edukacyjnych.



Rysunek 9.
Demonstracja działania algorytmu porządkowania przez wybór w programie **Sortowanie**

Opis algorytmu porządkowania przez wybór

Przedstawmy teraz ścisły opis algorytmu porządkowania przez wybór, znanego jako **SelectionSort**.

Algorytm porządkowania przez wybór – SelectionSort

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

Uwaga. Zauważ, że liczba k znaleziona w kroku 2 może być równa i w tym kroku, a zatem w kroku 3 ten sam element jest zamieniany miejscami ze sobą. Z taką sytuacja mamy do czynienia w piątej iteracji przykładowej demonstracji działania algorytmu, przedstawionej na rysunku 7, gdzie element 4 jest zamieniany ze sobą.

Złożoność algorytmu SelectionSort

Obliczmy teraz, ile porównań i zamian elementów, w zależności od liczby elementów w ciągu n, jest wykonywanych w algorytmie **SelectionSort** oraz w jego komputerowych implementacjach. W tym celu wystarczy zauważyć, o czym pisaliśmy już powyżej, że 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 już, że

algorytm znajdowania minimum w ciągu wykonuje o jedno porównanie mniej niż jest elementów w ciągu. Ponieważ w każdym kroku liczba elementów w przeszukiwanym podciągu jest o jeden mniejsza, cały algorytm porządkowania przez wybór, dla ciągu danych złożonego na początku z n elementów wykonuje liczbę porównań równą:

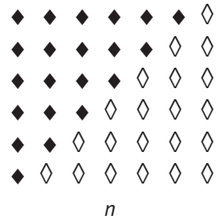
$$(n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$$

Wartość tej sumy można obliczyć wieloma sposobami. Przedstawimy dwa z nich – są one ciekawe przez swoją prostotę. Inny sposób, w którym korzysta się ze wzoru na sumę postępu arytmetycznego, pomijamy tutaj, jako oczywisty dla tych, którzy wiedzą, co to jest postęp arytmetyczny – nasze sposoby tego nie wymagają.

Dowód geometryczny

Kolejne liczby naturalne od 1 do $n-1$ można przedstawić w postaci trójkąta, którego wiersz i, licząc od dołu, zawiera i diamentów (na rys. 10 są to czarne diamenty). Dwa takie same trójkąty pasują do siebie i tworzą prostokąt zawierający $(n-1)n$ diamentów, zatem wartość powyższej sumy jest połową liczby wszystkich diamentów w całym prostokącie, czyli jest równa:

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



Ten geometryczny dowód, zamieszczony obok, znali już starożytni Grecy. Na podstawie geometrycznej interpretacji, wartość sumy kolejnych liczb naturalnych nazywali **liczbami trójkątnymi.**

Rysunek 10.
Ilustracja geometrycznego wyznaczania wartości sumy kolejnych liczb naturalnych od 1 do $n-1$

Spostrzeżenie nudzącego się geniusza

Geniusz – gen i już.
Hugo Steinhaus

Anegdota mówi, że nauczyciel matematyki w klasie, do której uczęszczał młody Carl Friedrich Gauss (1777-1855), jeden z największych matematyków w historii, by zająć przez dłuższy czas swoich uczniów żmudnymi rachunkami, dał im do obliczenia wartość sumy stu początkowych liczb naturalnych, czyli 1 + 2 + 3 + ... + 98 + 99 + 100. Nie cieszył się jednak zbyt długo spokojem, po chwili otrzymał bowiem gotową odpowiedź od Carla, który szybko zauważył, że suma liczb w skrajnych parach, 1+100, 2+99, 3+98 itd. aż do 50+51 jest taka sama, a takich par jest połowa ze stu, czyli z liczby wszystkich elementów. Stąd natychmiast otrzymał powyższy wzór.

Otrzymaliśmy wzór na liczbę porównań w algorytmie porządkowania przez wybór, który zależy tylko od liczby porządkowanych elementów. Można uznać za słabą stronę tego algorytmu, że wykonuje on taką samą liczbę działań (porównań i przestawień) na ciągach o tej samej długości, bez względu na stopień ich uporządkowania. W następnym podpunkcie wspominamy o metodzie porządkowania, która jest „bardzo czuła" na stopień uporządkowania elementów w porządkowanym ciągu.

4.4 INNE ALGORYTMY PORZĄDKOWANIA

Znanych jest bardzo wiele algorytmów porządkowania liczb i innych obiektów przechowywanych w komputerze, jak słów, dat (to nie są liczby tylko układy liczb), rekordów (czyli układów danych różnych typów). Temu zagadnieniu poświęcono wiele opasłych książek, np. Donald Knuth napisał na ten temat ponad 1000 stron

jeszcze w latach 60. XX wieku (patrz [4]). Na tych zajęciach problem sortowania pojawia się jeszcze kilka razy. W rozdziale 5 opisujemy algorytm porządkowania małych liczb, a w rozdziale 7 przedstawiamy algorytm sortowania przez scalanie bazując na metodzie dziel i zwyciężaj, będący jednym z najszybszych metod sortowania.

5 PORZĄDKOWANIE PRZEZ ZLICZANIE

W algorytmie porządkowania, który opisaliśmy w poprzednim rozdziale, nie poczyniliśmy żadnych założeń dotyczących elementów, które porządkujemy. Służy on do porządkowania dowolnych liczb (przyjęliśmy, że porządkowane liczby są całkowite), wykonując porównania między tymi liczbami – ważne jest tylko, która z dwóch liczby jest większa, a która mniejsza. W rzeczywistych sytuacjach porządkowane mogą być liczby szczególnej postaci, a ogólnie – porządkowane są najróżniejsze obiekty, np. słowa (do słownika), adresy (do książki telefonicznej) czy bardzo złożone zestawy danych, jak np. informacje o koncie bankowym i jego właścicielu.

Szczególnym przypadkiem porządkowania liczb jest sytuacja, w której liczby są niewielkie, jest ich niewiele różnych i należą do niewielkiego przedziału. Takie własności ma na przykład ciąg stopni wierzchołków w grafie (będzie o tym mowa na innych zajęciach). Na potrzeby tego rozdziału zmieńmy więc nieco opis problemu porządkowania, jak następuje:

Problem porządkowania niewielkich liczb

Dane: Liczba naturalna n i ciąg n liczb całkowitych $x_1, x_2, \ldots, x_n$, należących do przedziału $[1..M]$, gdzie M jest porównywalne co do wartości z n (na ogół przyjmuje się, że $M < n$).
Wynik: Uporządkowanie tego ciągu liczb od najmniejszej do największej.

Aby uporządkować liczby, które spełniają warunek z opisu danych, wystarczy policzyć, ile wśród danych jest liczb równych 1, liczby równych 2 itd. A następnie po policzeniu, ile jest konkretnych liczb, ustawić je w kolejności: najpierw elementy równe 1, później elementy równe 2 itd. Taki algorytm nazywa się **porządkowaniem przez zliczanie** (ang. *counting sort*). Załóżmy, że c_i oznacza liczbę elementów porządkowanego ciągu równych i – c_i można więc uznać za **licznik** elementów równych i. Ten algorytm składa się z dwóch kroków (krok 1 jest tylko przygotowaniem do porządkowania):

Algorytm. Porządkowanie przez zliczanie – CountingSort

Krok 1. Dla $i = 1, 2, \ldots, M$, przyjmij $c_i = 0$. {Zerowanie liczników elementów.}
Krok 2. Dla $i = 1, 2, \ldots, n$, zwiększ c_k o 1, gdzie $k = x_i$.
Krok 3. Zacznij od pierwszej pozycji w ciągu x i
dla $i = 1, 2, \ldots, M$, na kolejnych c_i pozycjach w ciągu x ustaw element i.

Z postaci algorytmu wynika, że jeśli porządkowane elementy przyjmują nie więcej niż M wartości i M nie jest większe od liczby porządkowanych elementów ($M < n$), to algorytm porządkowania przez zliczanie jest algorytmem o złożoności liniowej względem liczby elementów w porządkowanym ciągu. A zatem jest znacznie szybszy niż algorytm porządkowania przez wybór, przedstawiony w rozdziale 4.

6 POSZUKIWANIE INFORMACJI W ZBIORZE

Komputery bardzo ułatwiają szybkie poszukiwanie informacji umieszczonych na płytach CD i na serwerach sieci Internet. Jest to zasługą nie tylko szybkości działania procesorów, ale też dobrej organizacji pracy, dzięki czemu pozostaje im... niewiele do roboty. Przypomnijmy poczynione założenia, dotyczące przeszukiwanych

zbiorów. Zbiór może zawierać elementy powtarzające się (czyli o takich samych wartościach) i w obliczeniach jest dany w postaci ciągu, który na ogół jest poprzedzony liczbą jego elementów. Ciąg ten może być uporządkowany, np. od najmniejszego do największego elementu, lub nieuporządkowany.

W drugim rozdziale 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 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.

6.1 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 szczególnym przypadku może go tam nie być. W takim przypadku stosujemy **przeszukiwanie** (lub **poszukiwanie**) **liniowe**, które stosowaliśmy w rozdziale 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

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.
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.

Algorytm poszukiwania może wykonywać różną liczbę iteracji nawet dla ustalonego ciągu danych, zależy ona bowiem od wartości poszukiwanego elementu *y*. Mamy więc okazję, by posłużyć się instrukcją iteracyjną, w której liczba iteracji może zależeć od spełnienia podanego warunku. W przypadku algorytmu poszukiwania, kończy on działanie w jednej z dwóch sytuacji: albo został znaleziony poszukiwany element *y*, albo został przejrzany cały ciąg i nie znaleziono tego elementu. Musimy uwzględnić jedną i drugą ewentualność. Umożliwia nam to następująca funkcja – przyjmujemy, że wartością funkcji jest indeks znalezionego elementu, jeśli znajduje się on w ciągu, lub – 1, jeśli element o wartości *y* nie istnieje w ciągu:


```
function PrzeszukiwanieLiniowe(n,y:integer; x:tablicax):integer;
 {Wartoscia funkcji jest indeks elementu tablicy
  rownego y, lub -1, jesli brak takiego elementu w ciągu.}
 var i:integer;
begin
 i:=1;
 while (x[i]<>y) and (i<n) do i:=i+1;
 if x[i]=y then PrzeszukiwanieLiniowe:=i
 else PrzeszukiwanieLiniowe:=-1
end; {PrzeszukiwanieLiniowe}
```

Warunkowa instrukcja iteracyjna: `while (x[i]<>y) and (i<n) do i:=i+1;`

jest wykonywana tak długo, jak długo spełniony jest warunek: `(x[i]<>y) and (i<n)`

Czyli, gdy badany element ciągu jest różny od *y* oraz nie został jeszcze osiągnięty koniec ciągu. Wtedy zwiększany jest bieżący indeks elementów ciągu. Po tej instrukcji następuje złożona instrukcja warunkowa:

```
if x[i]=y then PrzeszukiwanieLiniowe:=i
else PrzeszukiwanieLiniowe:=-1
```

której zadaniem jest zbadanie, z jakiego powodu nastąpiło zakończenie iteracji. Jeśli `x[i]=y`, to w ciągu został znaleziony element równy `y`, a w przeciwnym razie (`else`) nie ma w ciągu elementu o wartości `y`.

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 – wartownika – właśnie o wartości *y*. W efekcie, przeszukiwanie zawsze zakończy się znalezieniem elementu o wartości *y*, należy jedynie sprawdzić, czy znaleziony element *y* 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. A oto fragment przeszukiwania z wartownikiem:

```
begin
 i:=1;
 x[n+1]:=y;
 while x[i]<>y do i:=i+1;
 if i<=n then PrzeszukiwanieLinioweWartownik:=i
 else PrzeszukiwanieLinioweWartownik:=-1
end;
```

6.2 POSZUKIWANIE ELEMENTU W ZBIORZE UPORZĄDKOWANYM

W tym podrozdziale zakładać będziemy, że poszukiwania elementów (informacji) są prowadzone w uporządkowanych zbiorach (ciągach) elementów – chcemy albo znaleźć element, albo umieścić go w takim zbiorze 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. Trzeba wyraźnie podkreślić, że:

integralną częścią informacji jest jej uporządkowanie,

gdyż w przeciwnym razie... nie jest to informacja. To stwierdzenie nie jest naukowym określeniem informacji[17], ale odnosi się do informacji w potocznym znaczeniu, do informacji, które nas zalewają i nieraz przytłaczają, do informacji, wśród których mamy odnaleźć tę nam potrzebną lub „zrobić wśród nich porządek". Podstawowym przygotowaniem do życia w erze i społeczeństwie informacji jest bowiem nabycie umiejętności takiego postępowania z informacją (uporządkowaną oczywiście), by w posługiwaniu się nią korzystać z jej uporządkowania, nie psuć go i ewentualnie naprawiać, gdy ulega zniszczeniu lub gdy informacja się rozrasta.

Wykonaj teraz ćwiczenie, które zapewne przeprowadziłeś 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*, *l* oraz *w* i znajdź je w wybranej książce. Zanotuj, ile razy ją otwierałeś, 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!

Pamiętaj. Staraj się wykorzystywać uporządkowanie zbioru, który przeszukujesz – znacznie skraca to Twoją pracę. Utrzymuj również porządek w swoich zbiorach (rzeczach), a będziesz mniej czasu tracił na ich przeszukiwanie.

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

Zabawa w zgadywanie liczb

Strategię podobną do poszukiwania w alfabetycznych spisach stosuje się podczas gry, polegającej na zgadywania ukrytej przez drugą osobę liczby. Wybierz sobie partnera do gry, która polega na odgadywaniu liczby naturalnej wybranej z przedziału [*m*, *n*]. Partner wybiera liczbę, a Ty masz ją odgadnąć. Na Twój wybór partner

[17] W teorii informacji, informacja jest definiowana jako „miara niepewności zajścia pewnego zdarzenia spośród skończonego zbioru zdarzeń możliwych" – na podstawie *Nowej encyklopedii powszechnej PWN*, t. 1-6, WN PWN, Warszawa 1996.

może jedynie odpowiedzieć: „tak", „za mała" lub „za duża". Jaką przyjmiesz strategię odgadywania liczby, by ją znaleźć w możliwie najmniejszej liczbie prób? Zamieńcie się rolami i powtórzcie te grę dla różnych przedziałów i dla różnych ukrytych liczb.

Rozegraj ze swoim partnerem kilka rund tej gry. Wśród ukrywanych liczb niech będą liczby z obu końców przedziału oraz liczba ze środka przedziału. Za lewy koniec przedziału wybierz również liczbę większą od 1.

Jeśli nie od razu, to na pewno po kilku próbach odkryjesz, że najlepsza strategia polega na podawaniu środkowych liczb z przedziału, w którym znajduje się poszukiwana liczba. Przypuśćmy, że poszukujemy liczby w przedziale [1, 100]. Jeśli pierwszym wyborem byłaby liczba 75 i otrzymalibyśmy odpowiedź „za duża", to pozostałby do przeszukania przedział [1, 75]. Jeśli natomiast wybierzemy w pierwszej próbie 50, to bez względu na to, jaką liczbę wybrał partner, pozostanie do przeszukania nie więcej niż pięćdziesiąt liczb, w przedziale [1, 49] albo [51, 100].

Przedstawione przykłady 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**. Opiszemy teraz ten algorytm dla trochę ogólniejszej sytuacji niż w zabawie w odgadywanie liczb. Po pierwsze zauważmy, że w podanej wyżej metodzie nie mają znaczenia wartości elementów w tablicy, tak długo, jak długo są uporządkowane. Musimy mieć jedynie pewność, że po porównaniu wskazanej w tablicy liczby z poszukiwaną i po wybraniu połowy zbioru, poszukiwana liczba znajduje się w wybranej połowie, a do tego wystarczy, by elementy w tablicy były uporządkowane, nie muszą być koniecznie kolejnymi liczbami. Tablica może również zawierać takie same liczby – wtedy oczywiście zajmują one miejsca obok siebie. Po drugie, poszukiwana liczba nie musi znajdować się w tablicy – wtedy naszą odpowiedzią będzie jakaś specjalnie wybrana liczba, np. –1.

Przyjmijmy, że przeszukiwany ciąg liczb jest umieszczony w tablicy $x[k..l]$. Załóżmy dodatkowo, 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, podobnie jak w grze w odgadywanie liczby, 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.

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 $x_s < y$, to *lewy* :=*s* + 1, a w przeciwnym razie *prawy* := *s* – 1.
Wróć do kroku 2.

Implementacja poszukiwania przez połowienie

Jak podaje Donald E. Knuth [4], napisanie w pełni poprawnej komputerowej implementacji algorytmu poszukiwania przez połowienie, sprawiło kłopot wielu programistom. Oto nasza implementacja:

```
function PrzeszukiwanieBinarne(x:tablicax; k,l:integer;
                               y:integer):integer;
  {Przeszukiwanie binarne ciagu x[k..l] w poszukiwaniu elementu y.
   Wartoscia funkcji jest indeks elementu tablicy rownego y, lub -1,
   jesli brak takiego elementu. W programie glownym nalezy zdefiniowac
   typ danych:  tablicax=array[1..n] of integer.}
 var Lewy,Prawy,Srodek:integer;
begin
 Lewy:=k;  Prawy:=l;
 while Lewy<=Prawy do begin
  Srodek:=(Lewy+Prawy) div 2;
  if x[Srodek]=y then begin
   PrzeszukiwanieBinarne:=Srodek; exit
  end;  {Element y nalezy do przeszukiwanego ciagu.}
  if x[Srodek]<y then Lewy:=Srodek+1
  else Prawy:=Srodek-1
 end;
 PrzeszukiwanieBinarne:=-1
end; {PrzeszukiwanieBinarne}
```

Złożoność algorytmu binarnego przeszukiwania

Nasuwa się teraz pytanie, ile porównań jest wykonywanych w algorytmie binarnego przeszukiwania. Załóżmy, że poszukiwany element znajduje się w ciągu – bo jeśli go tam nie ma, to jest wykonywana jedna dodatkowa iteracja (przekonaj się o tym).

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. Jak tę obserwację można uogólnić? Zapewne jest to związane z potęgą liczby 2, a dokładniej z najmniejszym wykładnikiem potęgi, której wartość nie jest mniejsza od *n*. 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**.

Poszukiwanie interpolacyjne, czyli poszukiwania w słownikach

Czy rzeczywiście przeszukiwanie binarne jest najszybszą metodą znajdowania elementu w zbiorze 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. 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 położenie szukanej liczby względem środka ciągu, ale uwzględnia jej wartość względem rozpiętości krańcowych wartości w ciągu. Szczegółowe informacje na temat poszukiwania interpolacyjnego można znaleźć w jednej z książek autora [7].

7 DZIEL I ZWYCIĘŻAJ, REKURENCJA – SORTOWANIE PRZEZ SCALANIE

W rozdziale 3 rozważaliśmy problem jednoczesnego znajdowania w zbiorze największego i najmniejszego elementu i podaliśmy algorytm **Max-i-Min**, w którym można wyróżnić trzy etapy:

1. Podział problemu na podproblemy.
2. Rozwiązywanie podproblemów.
3. Połączenie rozwiązań podproblemów w rozwiązanie głównego problemu.

Pierwszy etap polega na podziale zbioru danych na dwa podzbiory: kandydatów na maksimum i kandydatów na minimum, w drugim etapie – maksimum i minimum są znajdowane w zbiorach kandydatów, a trzeci etap w przypadku tego problemu jest jedynie wyprowadzeniem rozwiązania składającego się z dwóch liczb, otrzymanych jako rozwiązania dwóch podproblemów.

Podkreślaliśmy już wielokrotnie, że naturalnym podejściem w rozwiązywaniu problemów powinno być dążenie do wykorzystania znanych rozwiązań problemów – miejscem ku temu w powyższym schemacie jest etap drugi. W przypadku problemu **Max-i-Min**, korzystamy na tym etapie ze znanych algorytmów znajdowania największego i najmniejszego elementu w zbiorze. To naturalne podejście w rozwiązywaniu problemów przyjmuje szczególną postać, gdy tworzone podproblemy rozwiązujemy… tą samą metodą, jaką stosujemy do głównego problemu. A zatem, podproblemy również dzielimy na ich podproblemy i stosujemy… tę samą metodę. Kontynuujemy ten proces podziału tak długo, aż dojdziemy do podproblemu, dla którego znamy rozwiązanie, np. gdy ma mało elementów. W takich metodach rozwiązywania w naturalny sposób pojawia się **rekurencja**, która polega na tym, że algorytm odwołuje się do siebie samego. Na rekurencyjną metodę rozwiązywania problemów można więc spojrzeć również jak na chęć skorzystania ze znanego rozwiązania problemu, tylko że w tym przypadku jest to ten sam problem, a jego rozwiązanie… właśnie otrzymujemy.

> Nazwa zasady **dziel i zwyciężaj** pochodzi od angielskich słów *divide and conquer*. Nie należy jej jednak mylić z podobnie brzmiącą starożytną zasadą **dziel i rządź** (łac. *divide et impera*), która odnosiła się do sposobu rządzenia Cesarstwem Rzymskim, polegającego na dzieleniu wielkich obszarów i społeczeństw na mniejsze części, które w ten sposób miały utrudnioną komunikację między sobą, stanowiły więc mniejsze zagrożenie dla cesarzy. W przypadku zaś zasady **dziel i zwyciężaj** celem jest taki podział problemu na mniejsze części, by ich rozwiązania złożyły się na jak najefektywniejsze rozwiązanie głównego problemu.

Metoda **dziel i zwyciężaj** realizowana rekurencyjnie wraz z jej komputerową implementacją jest jedną z najsilniejszych technik komputerowego rozwiązywania problemów. W tej metodzie część organizacji obliczeń jest „zrzucana na komputer", faktycznie więc wykorzystywana jest potęga komputerów.

Zwróćmy jeszcze uwagę na bardzo ważną cechę metody dziel i zwyciężaj, dzięki której algorytmy tego typu są bardzo efektywne. Otóż na danym etapie podziału problemu na podproblemy staramy się, by problemy były definiowane na równolicznych lub niemal równolicznych podzbiorach danych. Jeśli więc dzielimy problem na

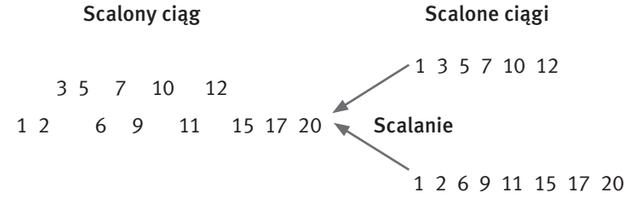
dwa podproblemy, to zwykle te podproblemy są definiowane na połowach zbioru danych. Ta własność nazywa się **równoważeniem** podproblemów.

Jeszcze jeden komentarz w kwestii, na ile podproblemów rozpada się problem w metodzie dziel i zwyciężaj. Najczęściej problem jest dzielony na dwa podproblemy i oba są dalej rozwiązywane. Nie zawsze tak musi być. Przeszukiwanie zbioru uporządkowanego metodą przez połowienie (p. 6.2) jest przykładem algorytmu dziel i zwyciężaj, w którym problem jest dzielony na dwa podproblemy ale dalej jest rozwiązywany tylko jeden z nich – w tej części danych, gdzie może znajdować się poszukiwany element. Znane są zastosowania metody dziel i zwyciężaj – na przykład mnożenie macierzy – w których problem o rozmiarze n jest dzielony na osiem podproblemów o rozmiarze $n/2$.

W tym rozdziale zastosujemy metodę dziel i zwyciężaj w algorytmie porządkowania przez scalanie. W tym algorytmie wykorzystywana jest **metoda scalania** uporządkowanych ciągów, czyli ich łączenia w jeden ciąg.

Scalanie ciągów uporządkowanych

Przyjmijmy, że scalane zbiory są uporządkowanymi ciągami liczb i wynik scalania ma być również ciągiem uporządkowanym. Scalenie dwóch uporządkowanych ciągów w jeden ciąg uporządkowany może być wykonane 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. Ilustrujemy to przykładem na rysunku 11.

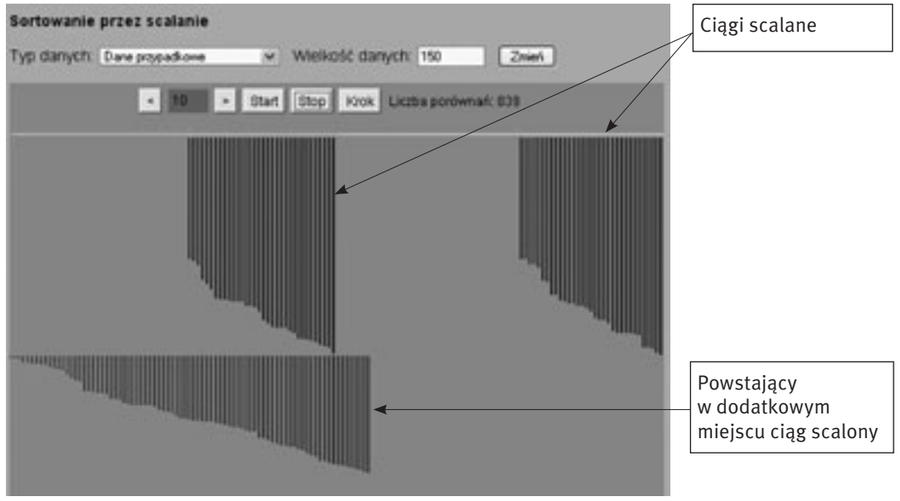


Rysunek 11.
Przykład scalania dwóch ciągów uporządkowanych

Kolejność przenoszenia elementów z ciągów zależy od ich wartości, które mogą powodować, że jeden ciąg może być znacznie wcześniej przeniesiony niż drugi. Czy można określić, ile porównań należy wykonać, aby scalić dwa ciągi? Liczba porównań może zależeć od wartości elementów. Zastanówmy się więc, ile najwięcej porównań musimy wykonać, by scalić dwa ciągi? Rozważanie różnych możliwych układów wartości elementów w obu scalanych ciągach nie daje szybkiej odpowiedzi na to pytanie. Spójrzmy natomiast od strony tworzonego ciągu. Z wyjątkiem elementu przeniesionego do niego 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 i każde z tych porównań jest związane z przeniesieniem jednego elementu, z wyjątkiem elementu, który trafia do scalonego ciągu na końcu.

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 *in situ*. Ta dodatkowo wykorzystywana pamięć jest słabą stroną operacji scalania.

Polecamy uruchomienie programu **Sortowanie** i przyjrzenie się rozszerzonej demonstracji działania algorytmu porządkowania przez scalanie. Zaobserwuj przebieg etapu scalania, który ma miejsce poniżej sortowanego ciągu, na kolejnym rysunku:



Rysunek 12.
Scalanie dwóch ciągów uporządkowanych – ilustracja z programu **Sortowanie** pokazująca, że scalanie nie może być wykonywane w tym samym miejscu

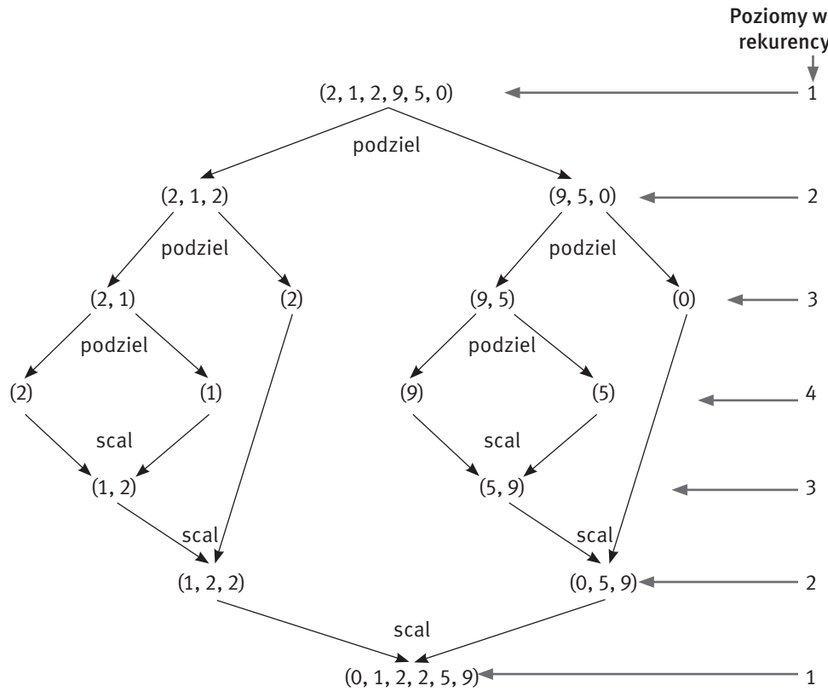
Algorytm scalania dwóch ciągów uporządkowanych – Scal

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 zastosować do tworzenia uporządkowanych ciągów z podciągów już uporządkowanych. A czy może być jakiś pożytek z tego algorytmu scalania uporządkowanych ciągów przy porządkowaniu dowolnych ciągów? Tak – jeśli potrafimy najpierw uporządkować te podciągi. A czy moglibyśmy założyć, że te podciągi zostały uporządkowane... tą samą metodą? Możemy – i tutaj przydaje się **rekurencja**. Dodatkowo założymy, że na każdym etapie podciągi mają prawie taką samą długość. Dochodzimy w ten sposób do metody porządkowania ciągu, która, na **zasadzie dziel i zwyciężaj**, scala dwa prawie równoliczne podciągi, uporządkowane tą samą metodą. Działanie tej metody można prześledzić na rysunku 13.

Zapiszemy teraz algorytm porządkowania przez scalanie w postaci listy kroków. Z powyższego szkicu oraz z ilustracji na rysunku 13 wynika, że 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 już bowiem ciąg uporządkowany. Zatem powrót z wywołań rekurencyjnych rozpoczyna się z ciągami złożonymi z pojedynczych elementów, które są scalane w ciągi o długości dwa, następnie w ciągi o długości trzy lub cztery itd. 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.

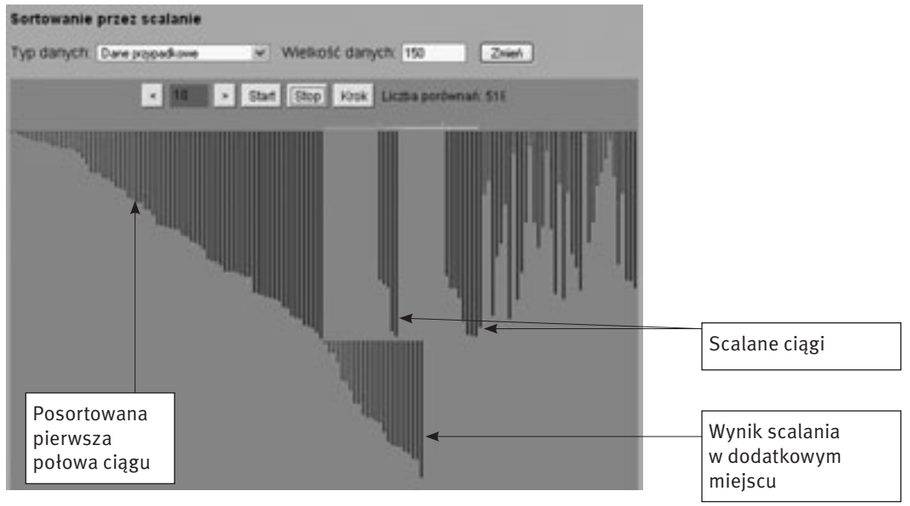


Rysunek 13.
Przykład działania algorytmu porządkowania przez scalanie

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 $(x_l, ..., x_p)$.

Obejrzyj na rysunku 14 kolejną demonstrację w programie **Sortowanie**. Zwróć uwagę na poszczególne etapy algorytmu: wywołania rekurencyjne dla coraz krótszych ciągów i scalanie podciągów w coraz dłuższe ciągi.

Jak już wspomnieliśmy przy okazji scalania ciągów – wykonanie tej operacji wymaga dodatkowego miejsca w pamięci komputera (patrz również rys. 12 i 14). Można jednak nie zapełniać tego miejsca w całości, oszczędzając przy tym na liczbie wykonywanych operacji. W algorytmie scalania dwóch uporządkowanych ciągów **Scal**, w kroku 2 końcowa część jednego z ciągów jest dopisywana do ciągu *z*. Gdy są scalane dwa podciągi tego samego ciągu (jak w algorytmie porządkowania przez scalanie), to nie zawsze jest to konieczne: jeśli należy przenieść końcową część pierwszego podciągu, to można od razu umieścić ją na końcu tworzonego ciągu (uwaga, trzeba to robić od końca przenoszonego podciągu), a jeśli należy przenieść końcową część drugiego podciągu, to można zostawić ją tam, gdzie jest. Taka modyfikacja kroku 4 w algorytmie **MergeSort** została uwzględniona w implementacji tego algorytmu na następnej stronie:



Rysunek 14.
Demonstracja działania porządkowania przez scalanie w programie **Sortowanie**. Posortowana jest już pierwsza połowa ciągu i w trakcie sortowania drugiej połowy scalane są dwa podciągi z pierwszej części drugiej połowy, uporządkowane wcześniej rekurencyjnie tą samą metodą

```
procedure MergeSort(Dol,Gora:integer; var x:Tablica1n);
 {Porzadkowanie metoda przez scalanie.}
var s:integer;
procedure Scal(l,s,p:integer);
  {Scalanie podciagow uporzadkowanych x[l..s-1] i x[s..p]
   w ciag uporzadkowany x[l..p].}
 var i,j,k,m:integer;
     z      :Tablica1n;
begin
 i:=l;  j:=s;  m:=l;
 while (i<s) and (j<=p) do begin
  if x[i]<=x[j] then begin
   z[m]:=x[i];  i:=i+1 end
  else begin z[m]:=x[j];  j:=j+1 end;
  m:=m+1
 end;  {while}
 if i<s then begin
  j:=s-1;  k:=p;
  while j>=i do begin
   x[k]:=x[j];
   k:=k-1;  j:=j-1
  end
 end;
 for i:=l to m-1 do x[i]:=z[i]
end;  {Scal}
begin {MergeSort}
```

```
  if Dol<Gora then begin
   s:=(Dol+Gora) div 2;
   MergeSort(Dol,s,x);
   MergeSort(s+1,Gora,x);
   Scal(Dol,s+1,Gora)
  end
 end; {MergeSort}
```

Złożoność sortowania ciągu n liczb przez scalanie wynosi około $n\log_2 n$, jest zatem znacznie mniejsza niż złożoność algorytmu sortowania przez wybór.

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. Nievergelt J., *Co to jest dydaktyka informatyki?*, „Komputer w Edukacji" 1/1994
6. Steinhaus H., *Kalejdoskop matematyczny*, WSiP, Warszawa 1989
7. Sysło M.M., *Algorytmy*, WSiP, Warszawa 1997
8. 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
9. Wirth N., *Algorytmy + struktury danych = programy*, WNT, Warszawa 1980