Marek Gągolewski
Instytut Badań Systemowych PAN
Wydział Matematyki i Nauk Informacyjnych Politechniki Warszawskiej

Algorytmy
i podstawy programowania

4. Wskaźniki i dynamiczna alokacja pamięci. Proste algorytmy sortowania tablic

Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

4.1. Dynamiczna alokacja pamięci

4.1.1. Organizacja pamięci komputera

W drugim rozdziale skryptu dowiedzieliśmy się, że w *pamięci operacyjnej* komputera przechowywane są nie tylko dane, ale i kod maszynowy programów. Podstawową jednostką pamięci jest *komórka* o rozmiarze jednego bajta. Każda komórka pamięci posiada swój *adres*, który jest reprezentowany we współczesnych komputerach za pomocą 32- lub 64-bitowej liczby całkowitej.

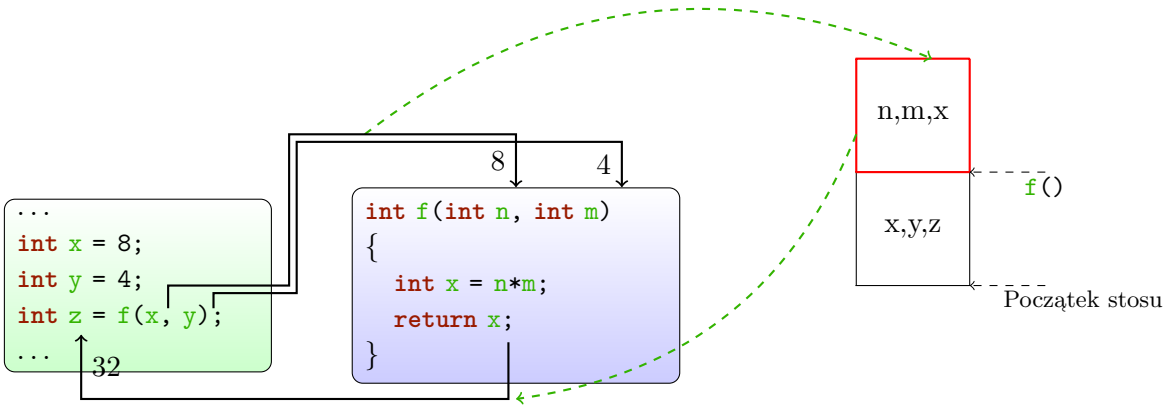
Z punktu widzenia każdego programu można wyróżnić następujący *podział puli adresowej* pamięci (w tzw. architekturze von Neumanna):

- kod programu — informacje interpretowane są tutaj jako instrukcje procesora,
- stos (ang. *stack*) — gdzie przechowywane są wartości zmiennych lokalnych funkcji,
- sterta (ang. *heap*) — gdzie znajdują się dane dynamicznie przydzielane (alokowane) na prośbę programu (zob. dalej),
- część niedostępna — zarządzana przez system operacyjny (m.in. dane innych programów).

Zatem każdy program przechowuje dane potrzebne do wykonywania swych czynności na *stosie* i *stercie*.

Stos jest częścią pamięci operacyjnej, na której dane umieszczane i kasowane są w porządku „ostatni na wejściu, pierwszy na wyjściu” (LIFO, ang. *last-in-first-out*). Umieszczanie i kasowanie danych na stosie odbywa się automatycznie. Każda wywoływana funkcja tworzy na stosie miejsce dla swoich zmiennych lokalnych. Gdy funkcja kończy działanie, usuwa z niego te informacje (to dlatego zmienne lokalne przestają wtedy istnieć).

Przyjrzyjmy się rozszerzonej wersji ilustracji z poprzedniego rozdziału (rys. 4.1). Po lewej stronie widzimy fragment funkcji `main()`, w której zostały zadeklarowane zmienne `x`, `y`, i `z`. Umieszczone są one na „dole” stosu (jako pierwsze w programie). Gdy funkcja ta wywołuje `f()`, na stosie tworzone jest miejsce dla zmiennych `n`, `m` i `x`. Gdy `f()` kończy swe działanie, są one ze stosu automatycznie usuwane.



Rys. 4.1. Zasięg zmiennych

4.1.2. Wskaźniki

Każda zmienna ma przyporządkowaną komórkę (bądź komórki) pamięci, w której przechowuje swoje dane, np. zmienna typu `int` zajmuje najczęściej 4 takie komórki (4 bajty). Fizyczny adres zmiennej (czyli numer komórki) można odczytać za pomocą operatora `&`.

```
int x;
cout << "x znajduje się pod adresem " << &x << endl;
// np. 0xe3d30dbc
```

Przypomnijmy, że 0xe3d30dbc oznacza liczbę całkowitą zapisaną w systemie szesnastkowym. W systemie dziesiętnym jest ona równa 3822259644. Co ważne, przy kolejnym uruchomieniu programu może to być inna wartość. Nas jednak interesuje tutaj fakt, że jest to „zwyczajna" liczba.

Każda zmienna ma zatem swoje „miejsce na mapie" (tzn. w pamięci komputera), znajdujące się pod pewnym adresem (np. na ul. Koszykowej 75 w Warszawie). Operator `&` pozwala więc uzyskać informację o pozycji danej zmiennej. Jeszcze inaczej: zmienna to „budynek magazynu" w którym można przechowywać towar określonego rodzaju, np. cukierki. Adres zmiennej to „współrzędne GPS" tegoż magazynu.

Specjalny typ danych do przechowywania informacji o adresach innych zmiennych („współrzędnych GPS") określonego typu zwany jest *typem wskaźnikowym*. Oznacza się go przez dodanie symbolu `*` (gwiazdka) bezpośrednio po nazwie typu.

Na przykład, zadeklarowanie zmiennej typu `int*` oznacza stworzenie zmiennej przechowującej fizyczny adres w pamięci komputera pewnej liczby całkowitej (współrzędne GPS pewnego magazynu do przechowywania cukierków), czyli *wskaźnika* na zmienną typu `int`.

Zapamiętaj

Istnieje specjalne miejsce w pamięci o adresie 0 (`NULL`, „czarna dziura"), do którego odwołanie się podczas działania programu powoduje wystąpienie błędu. Często używa się tego adresu np. do zainicjowania wskaźników celem oznaczenia, że początkowo nie wskazują one na „żadne konkretne miejsce".

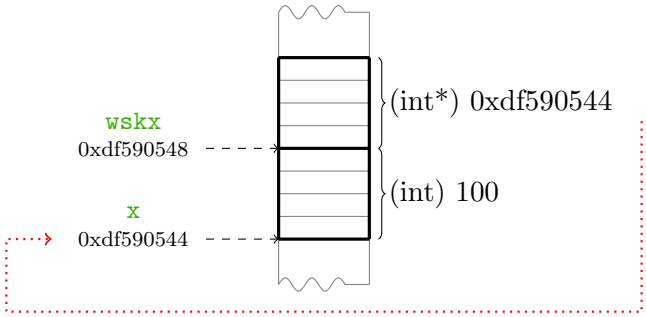
Na zmiennych wskaźnikowych został określony tzw. *operator wyłuskania*, `*` (nie mylić z „gwiazdką" modyfikującą znaczenie typu!), dzięki któremu możemy odczytać, co się znajduje pod danym adresem pamięci (co znajduje się w jakimś magazynie, którego znamy tylko współrzędne GPS).

Przyjrzyjmy się poniższemu przykładowi. Tworzone są dwie zmienne: jedna typu całkowitego, a druga wskaźnikowa. Ich rozmieszczenie w pamięci (a dokładniej na stosie, są to bowiem zmienne lokalne jakiejś funkcji) przedstawia rys. 4.2. Każda z tych zmiennych umieszczona jest pod jakimś adresem w pamięci RAM — można go odczytać za pomocą operatora `&`.

Listing 4.1. „Wyłuskanie" danych spod danego adresu

```
1 int  x = 100;
2 int* wskx = &x;
3
4 cout <<  wskx << endl; // np. 0xdf590544
5 cout << *wskx << endl; // 100
6 cout <<     x << endl; // 100
```

Wypisanie wartości wskaźnika oznacza wypisanie adresu, na który wskazuje. Wypisanie zaś „wyłuskanego" wskaźnika powoduje wydrukowanie wartości komórki pamięci, na którą pokazuje wskaźnik. Jako że zmienna typu `int` u nas ma rozmiar 4 bajtów, adres następnej zmiennej (`wskx`) jest o 4 jednostki większy od adresu `x`.



Rys. 4.2. Zawartość pamięci komputera w programie z listingu 4.1

Aby jeszcze lepiej zrozumieć omawiane zagadnienie, rozważmy fragment kolejnego programu.

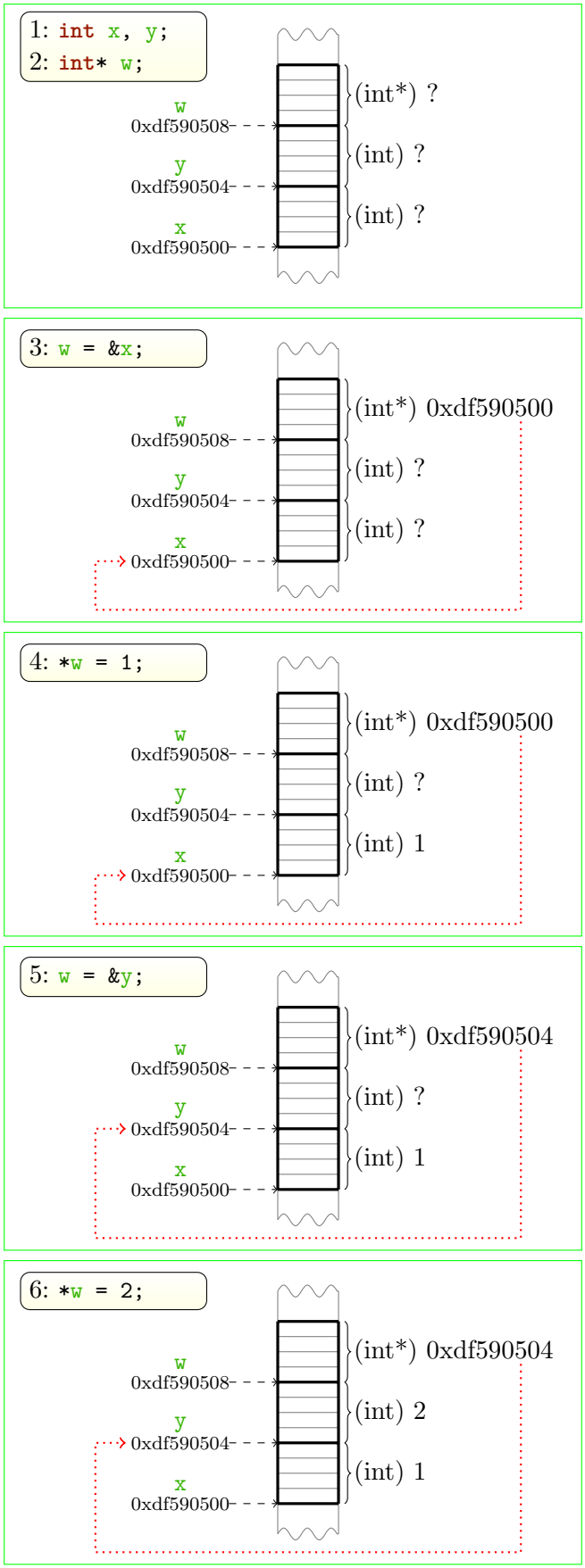
Listing 4.2. Proste operacje z użyciem wskaźników

```
1 int  x, y;
2 int* w;
3 w  = &x; // w = adres x (niech w wskazuje na zmienną x)
4 *w = 1;  // wstaw 1 tam, gdzie wskazuje teraz w
5 w  = &y; // w = adres y (niech w wskazuje na zmienną y)
6 *w = 2;  // wstaw 2 tam, gdzie wskazuje teraz w
```

Zawartość pamięci po wykonaniu kolejnych linii kodu przedstawia rys. 4.3. Tym razem za pomocą operatora wyłuskania zapisujemy dane do komórek pamięci, na które pokazuje wskaźnik `w`.

Zadanie

Prześledź pokazane rysunki bardzo uważnie. Wskaźniki są niezmiernie istotnym elementem języka C++.



Rys. 4.3. Zawartość pamięci po wykonaniu kolejnych instrukcji z listingu 4.2

Przykład z rozdz. 3 (cd.). W poprzednim rozdziale rozważaliśmy funkcję, która służyła do zamiany wartości dwóch zmiennych całkowitych. Przypomnijmy, że prawidłowe rozwiązanie tego problemu wymagało użycia nie argumentów przekazanych przez wartość, ale przez referencję. Równie skutecznym, acz w tym wypadku może nieco mniej eleganckim, jest użycie w tym przypadku wskaźników.

```
1  void zamien(int* x, int* y) {
2      int t = *x;
3      *x = *y;
4      *y = t;
5  }
6
7  int main() {
8      int n = 1, m = 2;
9      zamien(&n, &m); // przekazanie argumentów przez wskaźnik
10     cout << n << ", " << m << endl;
11     return 0;
12 }
```

Dzięki temu, że przekazaliśmy funkcji `zamien()` *adresy* zmiennych zadeklarowanych w funkcji `main()`, możemy nie tylko odczytywać, ale i nadpisywać tutaj ich wartości. Ominęliśmy tym samym ograniczenia przekazywania argumentów przez wartość (kopiowanie) — dostęp do „oryginalnych" zmiennych mamy tutaj w sposób *pośredni*, tj. za pomocą operatora wyłuskania `*`. ■

Ciekawostka

Pamiętamy, że aby dostać się do konkretnego pola struktury, należy użyć operatora „." (kropki). Jeśli mamy dostęp do wskaźnika na strukturę, możemy użyć do tego celu operatora „->".

```
struct Punkt {
    double x;
    double y;
};

// ... (np. main()) ...
Punkt p;
Punkt* wp = &p; // wskaźnik na p
wp->x = 1.0; // to samo, co (*wp).x = 1.0 - drugie mniej wygodne
wp->y = 2.0; // to samo, co (*wp).y = 2.0
// ...
```

4.1.3. Przydział i zwalnianie pamięci ze sterty

Oprócz ściśle określonej na etapie pisania programu ilości danych na stosie, można również dysponować pamięcią na *stercie*, por. s. 1. Miejsce na nasze dane może być przydzielane (alokowane) *dynamicznie* podczas działania programu za pomocą operatora `new`. Po użyciu należy je zwolnić za pomocą operatora `delete`.

Zapamiętaj

Zaalokowany obiekt będzie istniał w pamięci nawet po wyjściu z funkcji, w której go stworzyliśmy! Dlatego należy pamiętać, aby go usunąć w pewnym miejscu kodu.

Oto, w jaki sposób możemy dokonać alokacji i dealokacji pamięci dla jednego obiektu.

```
typ* obiekt = new typ; // alokacja (new zwraca wskaźnik na
    przydzielone miejsce w pamięci)
// ...
delete obiekt; // zwolnienie pamięci
```

Co bardzo ważne, możemy w ten sposób również przydzielić pamięć na wiele obiektów następujących kolejno po sobie.

```
int n = 4;
typ* obiekt = new typ[n]; // alokacja (zwraca wskaźnik na pierwszy
    z obiektów)
// ...
delete [] obiekt; // uwaga na ,,[]'' - wiele obiektów!
```

Za pomocą powyższego kodu utworzyliśmy *ciąg* obiektów określonego typu, czyli inaczej *tablicę*.

4.1.4. Tablice

Do tej pory przechowywaliśmy dane używając pojedynczych zmiennych. Były to tzw. *zmienne skalarne* (atomowe). Pojedyncza zmienna odpowiadała jednej „jednostce informacji" (liczbie, wartości logicznej, złożonej strukturze, wskaźnikowi).

Często jednak w naszych programach będzie zachodzić potrzeba rozważenia ciągu n zmiennych tego samego typu, gdzie n niekoniecznie musi być znane z góry. Dla przykładu, rozważmy fragment programu dokonujący podsumowania rocznych zarobków pewnego dość obrotnego studenta.

```
1  double zarobki1, zarobki2, /*...*/, zarobki12;
2                                     // deklaracja 12 zmiennych
3  zarobki1  = 1399.0; // styczeń
4  zarobki2  = 1493.0; // luty
5  // ...
6  zarobki12 = 999.99; // grudzień
7
8  double suma = 0.0;
9  suma += zarobki1;
10 suma += zarobki2;
11 //...
12 suma += zarobki12;
13
14 cout << "Zarobiłem w 2012 r. " << suma << " zł.";
```

Dochód z każdego miesiąca przechowywany jest w oddzielnej zmiennej. Nietrudno zauważyć, że operowanie na nich nie jest zbyt wygodne. Mało tego, dość żmudne byłoby rozszerzanie funkcjonalności takiego programu na przypadek obejmujący podsumowanie np. zarobków z 2,3,... lat.

Rozwiązanie tego problemu może być jednak bardzo czytelnie zapisane z użyciem dynamicznie alokowanych tablic, które są reprezentacją znanych nam obiektów matematycznych: *ciągów skończonych* bądź *wektorów*.

Wiemy, że za pomocą operatora `new` możemy przydzielić pamięć dla $n \geqslant 1$ obiektów określonego typu. Operator ten zwraca wskaźnik na pierwszy element takiego ciągu. Pozostaje tylko odpowiedzieć sobie na pytanie, w jaki sposób możemy się dostać do kolejnych elementów.

```
double* zarobki = new double[12];
// zarobki - wskaznik na pierwszy element ciągu
*zarobki = 1399.0; // wprowadź zarobki w pierwszym miesiącu
// co dalej?
delete [] zarobki; // zwolnienie pamięci
```

Przypomnijmy, wskaźnik jest po prawdzie liczbą całkowitą. Okazuje się, że została określona na nim operacja dodawania. I tak `zarobki+i`, gdzie `i` jest liczbą całkowitą nieujemną, oznacza „podaj adres `i`-tego obiektu z ciągu”. Tym samym `zarobki+0` jest tym samym, co po prostu `zarobki` (adresem pierwszego elementu), a `zarobki+n-1`, adresem ostatniego elementu z `n`-elementowego ciągu.

Wobec powyższego, fragment pierwotnej wersji programu związany z wprowadzeniem zarobków możemy zapisać w następujący sposób:

```
double* zarobki = new double[12];
// zarobki - wskaznik na pierwszy element ciągu
*(zarobki+0)  = 1399.0; // wprowadź zarobki w pierwszym miesiącu
*(zarobki+1)  = 1493.0; // wprowadź zarobki w drugim miesiącu
// ...
*(zarobki+11) = 999.99; // wprowadź zarobki w ostatnim miesiącu
// ...
delete [] zarobki; // zwolnienie pamięci
```

Zapamiętaj

Wygodniejszy dostęp do poszczególnych elementów tablicy możemy uzyskać za pomocą *operatora indeksowania*, „$[\cdot]$”.

Jeśli `t` jest tablicą (a ściślej: wskaźnikiem na pierwszy element ciągu obiektów przydzielonych dynamicznie), to `*(t+i)` możemy zapisać równoważnie przez `t[i]`.

Elementy tablicy są numerowane od 0 do $n-1$, gdzie n to rozmiar tablicy.

Operator indeksowania przyjmuje za argument dowolną wartość całkowitą (np. stałą bądź wyrażenie arytmetyczne). Każdy element tablicy traktujemy tak, jakby był zwykłą zmienną — taką, z którą do tej pory mieliśmy do czynienia.

Informacja

W języku C++ nie ma mechanizmów sprawdzania poprawności indeksów! Następujący kod być może (nie wiadomo) nie spowoduje błędu natychmiast po uruchomieniu.

```
1 int* t = new int[5];
2 t[-100]  = 15123;  // :-(
3 t[10000] = 25326;  // :-(
4 delete [] t;
```

Powyższe instrukcje jednak zmieniają wartości komórek pamięci reprezentujących dane innych obiektów. Skutki tego działania mogą się objawić w innym miejscu programu, powodując nieprzewidywalne i trudne do wykrycia błędy.

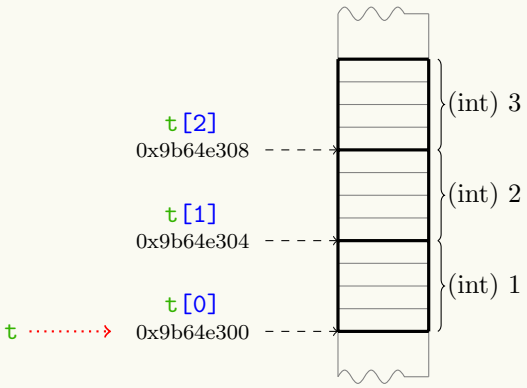
Ciekawostka

Niech `t` będzie wskaźnikiem na pewien `typ`. Zauważmy, że zapis `t+i` nie oznacza koniecznie, że chodzi nam o adres przechowywany w zmiennej wskaźnikowej `t` *plus* jeden bajt. Operacja dodawania bierze pod uwagę `typ` zmiennej wskaźnikowej i dokonuje „przesunięcia" adresu o wielokrotność liczby bajtów, które zajmuje w pamięci jedna zmienna typu `typ`.

Możemy to sprawdzić np. w następujący sposób.

```
int* t = new int[3]; // u nas int to 4 bajty
cout << t;     // np. 0x9b64e300 - to samo, co cout << &t[0];
cout << t+1;   // np. 0x9b64e304 - to samo, co cout << &t[1];
cout << t+2;   // np. 0x9b64e308 - to samo, co cout << &t[2];
t[0] = 1;
t[1] = 2;
t[2] = 3;
delete [] t;
```

Kolejne elementy tablicy w pamięci zawsze następują po sobie, co ilustruje poniższy rysunek.



Jesteśmy już gotowi, by napisać fragment programu do sumowania zarobków naszego kolegi. Tym razem nie będziemy zakładać, że liczba miesięcy jest określona z góry. Tutaj będziemy ją wprowadzać z klawiatury.

```
int n;
cout << "Ile miesięcy? ";
cin >> n;
assert(n>0); // wymaga <cassert>

double* zarobki = new double[n];

for (int i=0; i<n; ++i) {
   cout << "Podaj zarobki w miesiącu nr " << i << ": ";
   cin >> zarobki[i]; // to samo, co cin >> *(zarobki+i);
}

// tutaj możemy np. wygenerować ładne zestawienie
```

```
14 // wprowadzonych danych itp.
15 for (int i=0; i<n; ++i)
16    cout << i << ": " << zarobki[i] << endl;
17
18 // policzmy sumę:
19 double suma = 0.0;
20 for (int i=0; i<n; ++i)
21    suma += zarobki[i];
22
23 cout << "Zarobiłem w 2012 r. " << suma << " zł.";
24
25 delete [] zarobki;
```

Ciekawostka

W języku C++ można także deklarować tablice o ustalonym z góry, stałym rozmiarze (na stosie). Ich zaletą jest możliwość ustalenia wartości ich elementów podczas deklaracji, co może poprawiać czytelność programów demonstrujących implementacje pewnych szczególnych algorytmów.

```
1 int t[3] = {1, 2, 3}; // deklaracja tablicy i inicjowanie
      wartości
2 // t wciąż jest wskaźnikiem na pierwszy element:
3 // używamy go tak, jak w przypadku tablicy dynamicznie
      alokowanej
4 // ...
5 // nie stosujemy delete! (bo nie było new)
```

Jednak w przypadku dużej klasy problemów ich implementacja z użyciem tablic o zadanym rozmiarze wydaje się mało naturalna i słabo „skalowalna” względem rozmiaru problemu.

4.1.5. Przekazywanie tablic funkcjom

Podsumujmy: tablica to ciąg obiektów tego samego typu położonych w pamięci kolejno, jeden po drugim. Dostęp do elementów tablicy mamy zapewniony przez wskaźnik na pierwszy z tych obiektów. Obiekty typu wskaźnikowego możemy przekazywać funkcjom tak, jak zwykłe zmienne skalarne. Wobec tego, aby można było napisać funkcję, która w jakiś sposób przetwarza dane zawarte w tablicy, należy dodatkowo przekazać jej informację o tym, ile jest elementów w tablicy, czyli jej *rozmiar*.

Ciekawostka

W rozdziale 5 poznamy inny sposób informowania funkcji, gdzie znajduje się ostatni element tablicy. Założymy wtedy, że ostatnim elementem ciągu jest pewna specjalna wyróżniona wartość, (tzw. *wartownik*) która nie pojawia się jako „zwykły” element gdzie indziej. Wówczas wystarczy nam tylko informacja na temat położenia pierwszego elementu tablicy. Jej koniec bowiem będziemy mogli sami sobie znaleźć.

Przykład. *Funkcja wyznaczająca sumę wartości elementów z podanej tablicy.*

```
1  double suma(double const* t, int n)
2  {  // dostęp do elementów t tylko do odczytu
3     assert(n > 0);
4     double s = 0.0;
5     for (int i=0; i<n; ++i)
6        s += t[i];
7     return s;
8  }
9
10 int main(void)
11 {
12    int ile = 10;
13    double* punkty = new double[ile];
14    for (int i=0; i<ile; ++i)
15       cin >> punkty[i];
16    cout << suma(punkty, ile); // przekazanie tablicy do funkcji
17    delete [] punkty;
18    return 0;
19 }
```

Na marginesie, zauważmy, że funkcja `suma()` nie ma żadnych efektów ubocznych. Nie wypisuje nic na ekran ani o nic nie pyta się użytkownika. Wyznacza tylko wartość sumy elementów zawartych w tablicy — czyli tylko to, czego użytkownik (funkcja `main()`) może się po niej spodziewać.

Ponadto zapis `double const*` zapewnia, że elementów danej tablicy nie można zmieniać. Jest to tzw. wskaźnik na wartości stałe. ■

4.2. Proste algorytmy sortowania tablic

Rozważymy teraz problem *sortowania tablic jednowymiarowych*, który jest istotny w wielu zastosowaniach, zarówno teoretycznych jak i praktycznych. Dzięki odpowiedniemu uporządkowaniu elementów niektóre algorytmy (np. wyszukiwania) mogą działać szybciej, mogą być prostsze w napisaniu czy też można łatwiej formalnie udowodnić ich poprawność.

Zadanie

Wyobraź sobie, ile czasu zajęłoby Ci znalezienie hasła w słowniku (chodzi oczywiście o słownik książkowy), gdyby redaktorzy przyjęliby losową kolejność wyrazów. Przeanalizuj z jakiego „algorytmu” korzystasz wyszukując to, co Cię interesuje.

Co więcej, istnieje wcale niemało zagadnień, które wprost wymagają pewnego uporządkowania danych i które bez takiej operacji wcale nie mają sensu.

Zadanie

Przykładowo, rozważmy w jaki sposób wyszukujemy interesujące nas strony internetowe. Większość wyszukiwarek działa w następujący sposób.

1. Znajdź wszystkie strony w bazie danych, które zawierają podane przez użytkownika słowa kluczowe (np. „kaszel”, „gorączka” i „objawy”).
2. Oceń każdą znalezioną stronę pod względem pewnej miary adekwatności/popularności-/jakości.
3. Posortuj wyniki zgodnie z ocenami (od „najlepszej” do „najgorszej”) i pokaż ich listę użytkownikowi.

Warto przypomnieć, że o rynkowym sukcesie wyszukiwarki Google zadecydowało to, że — w przeciwieństwie do ówczesnych konkurencyjnych serwisów — zwracała ona wyniki w dość „pożytecznej” dla większości osób kolejności.

Problem sortowania, w swej najprostszej postaci, można sformalizować w następujący sposób.

> Dana jest tablica `t` rozmiaru `n` zawierająca elementy, które można porównywać za pomocą operatora relacyjnego `<=`. Należy zmienić kolejność (tj. dokonać permutacji, uporządkowania) elementów `t` tak, by zachodziły warunki:
>
> `t[0] <= t[1], t[1] <= t[2], ..., t[n−2] <= t[n−1].`

Ciekawostka

Zauważmy, że rozwiązanie takiego problemu wcale nie musi być jednoznaczne.

Dla tablic zawierających elementy `t[i]` i `t[j]` takie, że dla `i≠j` zachodzi `t[i] <= t[j]` oraz `t[j] <= t[i]`, tj. `t[i] == t[j]`, może istnieje więcej niż jedna permutacja spełniająca powyższe warunki.

Algorytm sortowania nazwiemy *stabilnym*, jeśli względna kolejność elementów o tej samej wartości zostaje zachowana po posortowaniu. Własność ta jest przydatna w przypadku sortowania obiektów złożonych za pomocą więcej niż jednego kryterium na raz.

W niniejszym paragrafie omówimy trzy algorytmy sortowania:

1. sortowanie przez wybór,
2. sortowanie przez wstawianie,
3. sortowanie bąbelkowe.

Algorytmy te cechują się tym, że w *pesymistycznym* („najgorszym”) przypadku liczba operacji porównań elementów tablicy jest *proporcjonalna* do n^2 (zob. dalej, podrozdz. 4.2.4). Bardziej wydajne i, co za tym idzie, bardziej złożone algorytmy sortowania będą omówione w semestrze III (np. sortowanie szybkie, przez łączenie, przez kopcowanie). Niektóre z nich wymagają co najwyżej $kn \log n$ porównań dla pewnego k. Dzięki temu dla tablic o dużym rozmiarze działają naprawdę szybko.

4.2.1. Sortowanie przez wybór

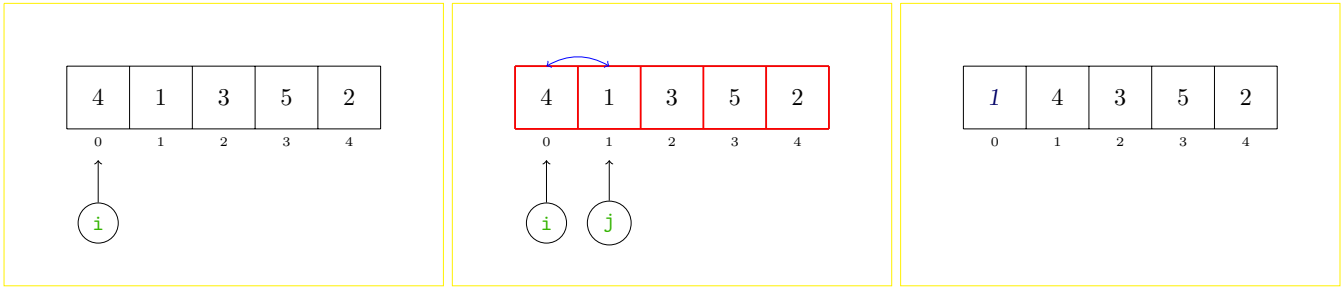
W algorytmie *sortowania przez wybór* (ang. *selection sort*) dokonujemy za każdym razem wyboru elementu najmniejszego spośród do tej pory nieposortowanych, póki cała tablica nie zostanie uporządkowana.

Ideę tę przedstawia następujący pseudokod:

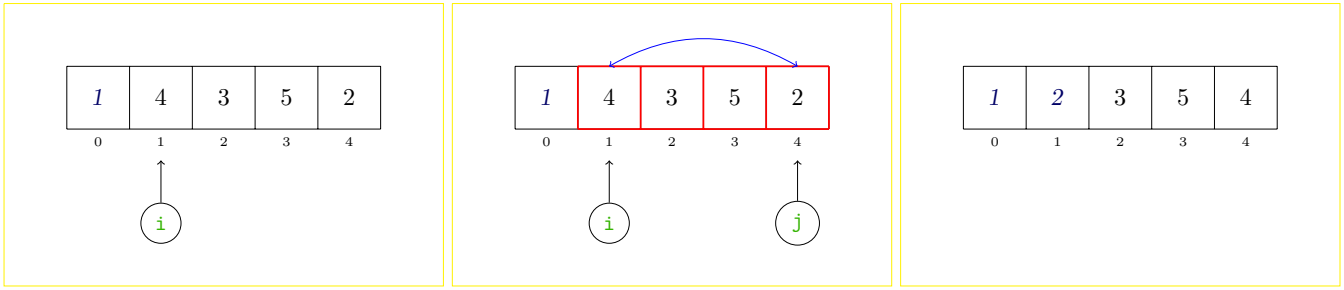
```
dla i=0,1,...,n-2
{
   // t[0], ..., t[i-1] są już uporządkowane względem relacji <=
   // (nadto, są już na swoich ostatecznych miejscach)
   j = indeks najmniejszego elementu
         spośród t[i], ..., t[n-1];
   zamień elementy t[i] i t[j];
}
```

Jako przykład rozważmy, krok po kroku, przebieg sortowania ciągu liczb naturalnych `(4,1,3,5,2)`. Kolejne iteracje działania tego algorytmu ilustrują rys. 4.4–4.8.

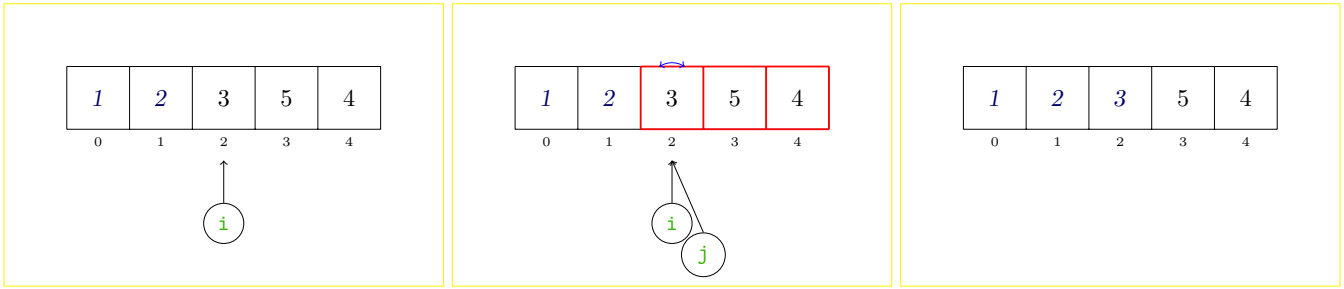
1. W kroku I (rys. 4.4) mamy `i==0`. Dokonując wyboru elementu najmniejszego spośród `t[0]`, ..., `t[4]` otrzymujemy `j==1`. Zamieniamy więc elementy `t[0]` i `t[1]` miejscami.
2. W kroku II (rys. 4.5) mamy `i==1`. Wybór najmniejszego elementu wśród `t[1]`, ..., `t[4]` daje `j==4` Zamieniamy miejscami zatem `t[1]` i `t[4]`.
3. Dalej (rys. 4.6), `i==2`. Elementem najmniejszym spośród `t[2]`, ..., `t[4]` jest `t[j]` dla `j==2` Zamieniamy miejscami zatem niezbyt sensownie `t[2]` i `t[2]`. Komputer, na szczęście, zrobi to bez grymasu.
4. W ostatnim kroku (rys. 4.7) `i==3` i `j==4`, dzięki czemu możemy uzyskać ostateczne rozwiązanie (rys. 4.8).



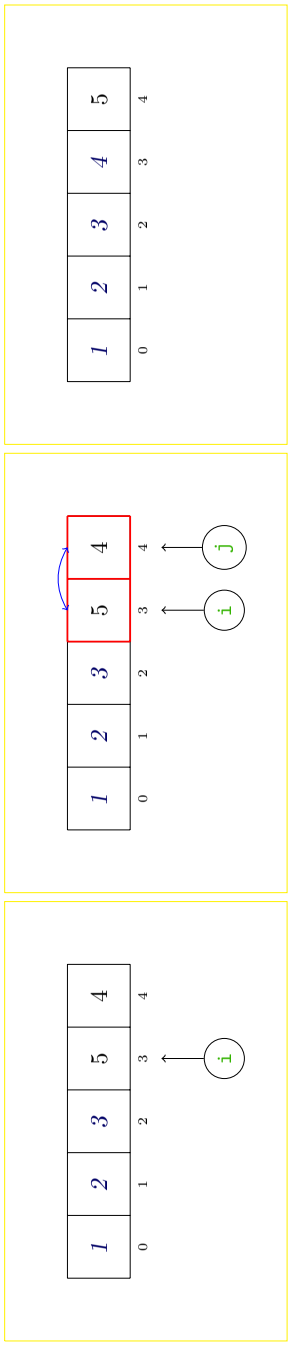
Rys. 4.4. Sortowanie przez wybór — przykład — iteracja I



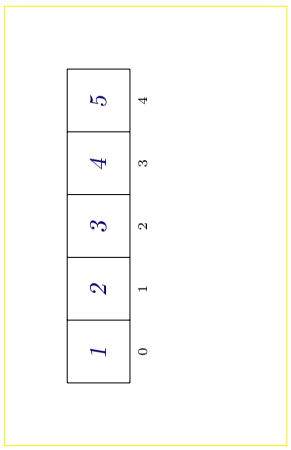
Rys. 4.5. Sortowanie przez wybór — przykład — iteracja II



Rys. 4.6. Sortowanie przez wybór — przykład — iteracja III



Rys. 4.7. Sortowanie przez wybór — przykład — iteracja IV



Rys. 4.8. Sortowanie przez wybór — przykład — rozwiązanie

4.2.2. Sortowanie przez wstawianie

Algorytm *sortowania przez wstawianie* (ang. *insertion sort*) jest metodą często stosowaną w praktyce do porządkowania małej liczby elementów (do ok. 20–30) ze względu na swą prostotę i szybkość działania.

W niniejszej metodzie w `i`-tym kroku elementy `t[0], ..., t[i-1]` są już wstępnie uporządkowane względem relacji `<=`. Pomiędzy nie wstawiamy `t[i]` tak, by nie zaburzyć porządku.

Formalnie rzecz ujmując, idea ta może być wyrażona za pomocą pseudokodu:

```
dla i=1,2,...,n-1
{
   // t[0], ..., t[i-1] są wstępnie uporządkowane względem <=
   // (ale niekoniecznie jest to ich ostateczne miejsce)
   j = indeks takiego elementu spośród t[0],...,t[i], że
      t[u] <= t[i] dla każdego u < j oraz
      t[i] <  t[v] dla każdego v ⩾ j;
   jeśli (j < i) wstaw t[i] przed t[j];
}
```

gdzie przez operację „`wstaw t[i] przed t[j]`”, dla $0 \leqslant j < i$ rozumiemy ciąg działań, mający na celu przestawienie kolejności elementów tablicy:

`t[0]`	...	`t[j-1]`	`t[j]`	...	`t[i-1]`	`t[i]`	`t[i+1]`	...	`t[n-1]`

tak, by uzyskać:

`t[0]`	...	`t[j-1]`	`t[i]`	`t[j]`	...	`t[i-1]`	`t[i+1]`	...	`t[n-1]`

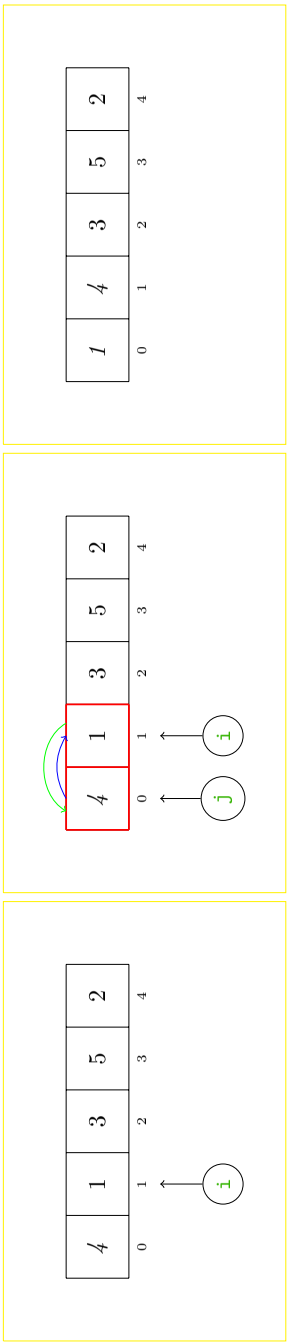
Powyższy pseudokod może być wyrażony w następującej równoważnej formie:

```
dla i=1,2,...,n-1
{
   // t[0], ..., t[i-1] są uporządkowane względem <=
   // (ale niekoniecznie jest to ich ostateczne miejsce)
   znajdź największe j ze zbioru {0,...,i} takie, że
      j == 0 lub t[j-1] <= t[i];
   jeśli (j < i) wstaw t[i] przed t[j];
}
```

Jako przykład rozpatrzmy znów ciąg liczb naturalnych `(4,1,3,5,2)`.
Przebieg kolejnych wykonywanych kroków przedstawiają rys. 4.9–4.12.

Ciekawostka

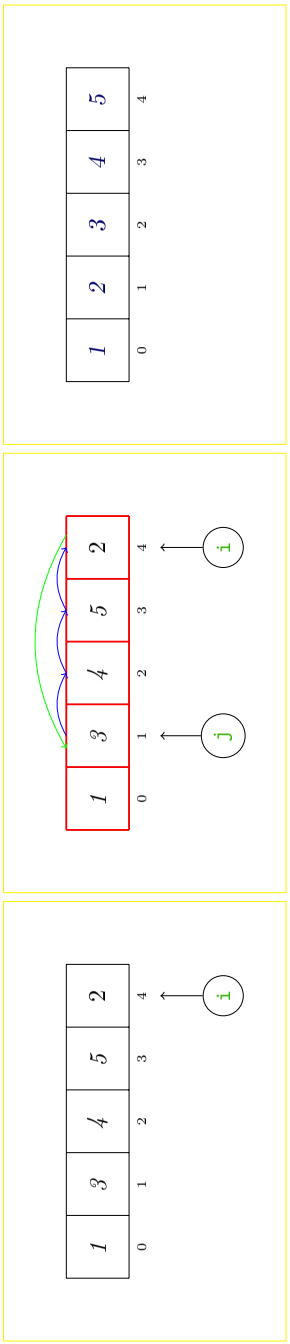
Można pokazać, że tak sformułowany algorytm jest stabilny. Prześledź jego działanie np. dla ciągu $(2', 4, 2'', 5, 1, 3)$ (dla czytelności te same elementy wyróżniliśmy, by wskazać ich pierwotny porządek). Porównaj uzyskany wynik z tym, który można uzyskać za pomocą algorytmu sortowania przez wybór.



Rys. 4.9. Sortowanie przez wstawianie — przykład — iteracja I

Rys. 4.10. Sortowanie przez wstawianie — przykład — iteracja II

Rys. 4.11. Sortowanie przez wstawianie — przykład — iteracja III



Rys. 4.12. Sortowanie przez wstawianie — przykład — iteracja IV

4.2.3. Sortowanie bąbelkowe

Sortowanie bąbelkowe (ang. *bubble sort*) jest interesującym przykładem algorytmu pojawiającego się w większości podręczników akademickich dotyczących podstawowych sposobów sortowania tablic, którego prawie wcale nie stosuje się w praktyce. Jego wydajność jest bowiem bardzo słaba w porównaniu do dwóch metod opisanych powyżej. Z drugiej strony, posiada on sympatyczną „hydrologiczną" (nautyczną?) interpretację, która urzeka wielu wykładowców, w tym i skromnego autora niniejszej książeczki. Tak umotywowani, przystąpmy więc do zapoznania się z nim.

W tym algorytmie porównywane są tylko elementy ze sobą bezpośrednio sąsiadujące. Jeśli okaże się, że nie zachowują one odpowiedniej kolejności względem relacji `<=`, element „cięższy" wypychany jest „w górę", niczym pęcherzyk powietrza (tytułowy bąbelek) pod powierzchnią wody.

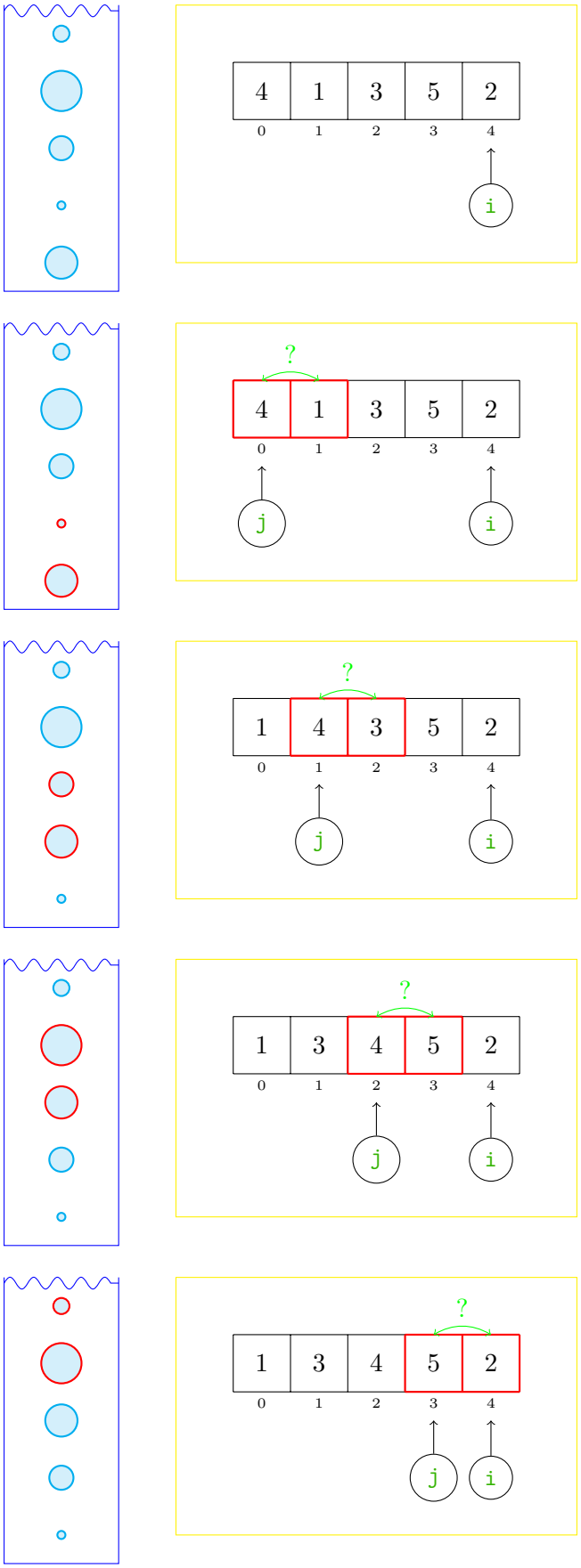
A oto pseudokod:

```
dla i=n-1,...,1
{
   dla j=0,...,i-1
   {
      // porównuj elementy parami
      jeśli (t[j] > t[j+1])
         zamień t[j] i t[j+1];
         // tzn. "wypchnij" cięższego "bąbelka" w górę
   }
   // tutaj elementy t[i],...,t[n-1] są już na swoich miejscach
}
```

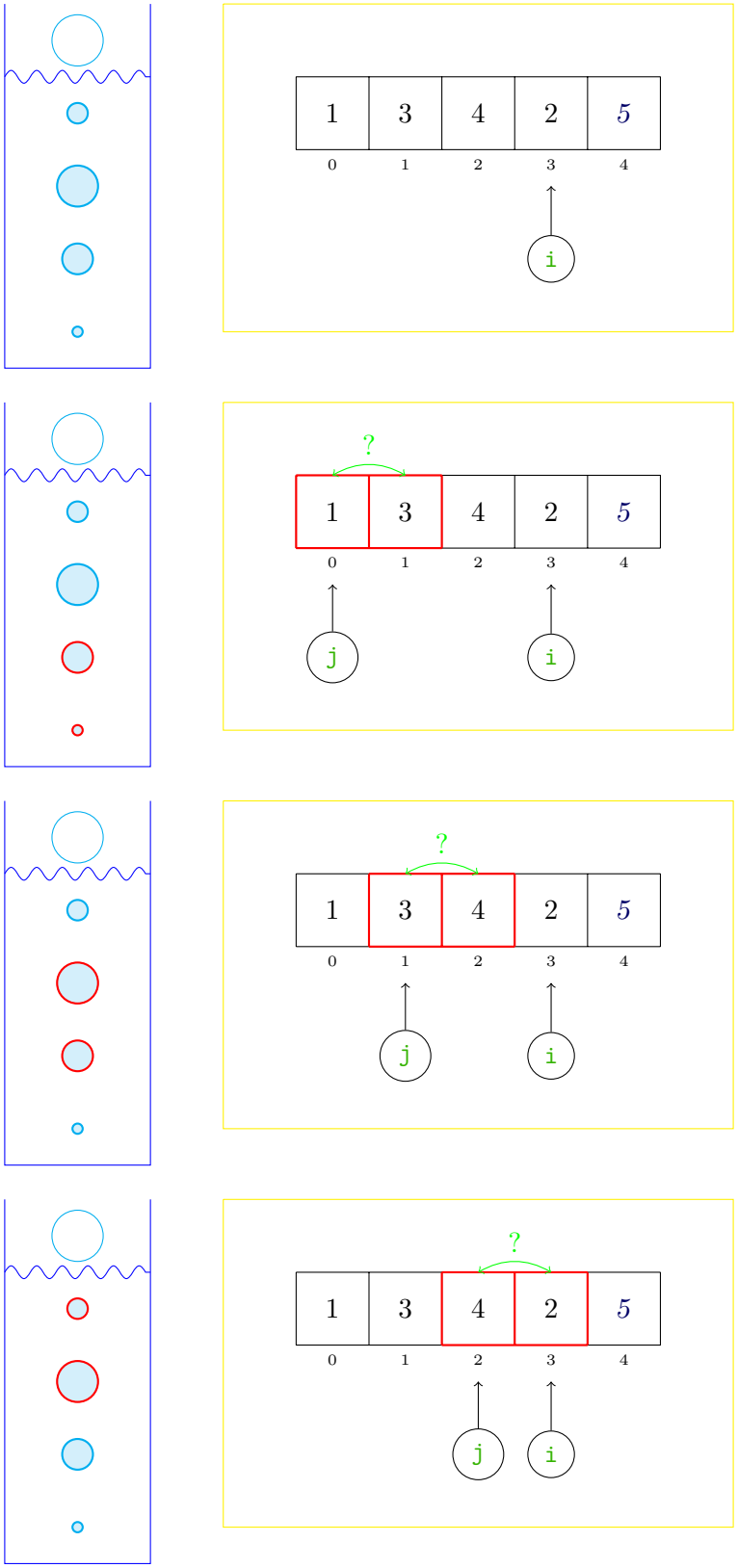
Dla przykładu rozpatrzmy ponownie tablicę `(4,1,3,5,2)`.
Przebieg kolejnych wykonywanych kroków przedstawiają rys. 4.13–4.17.

Ciekawostka

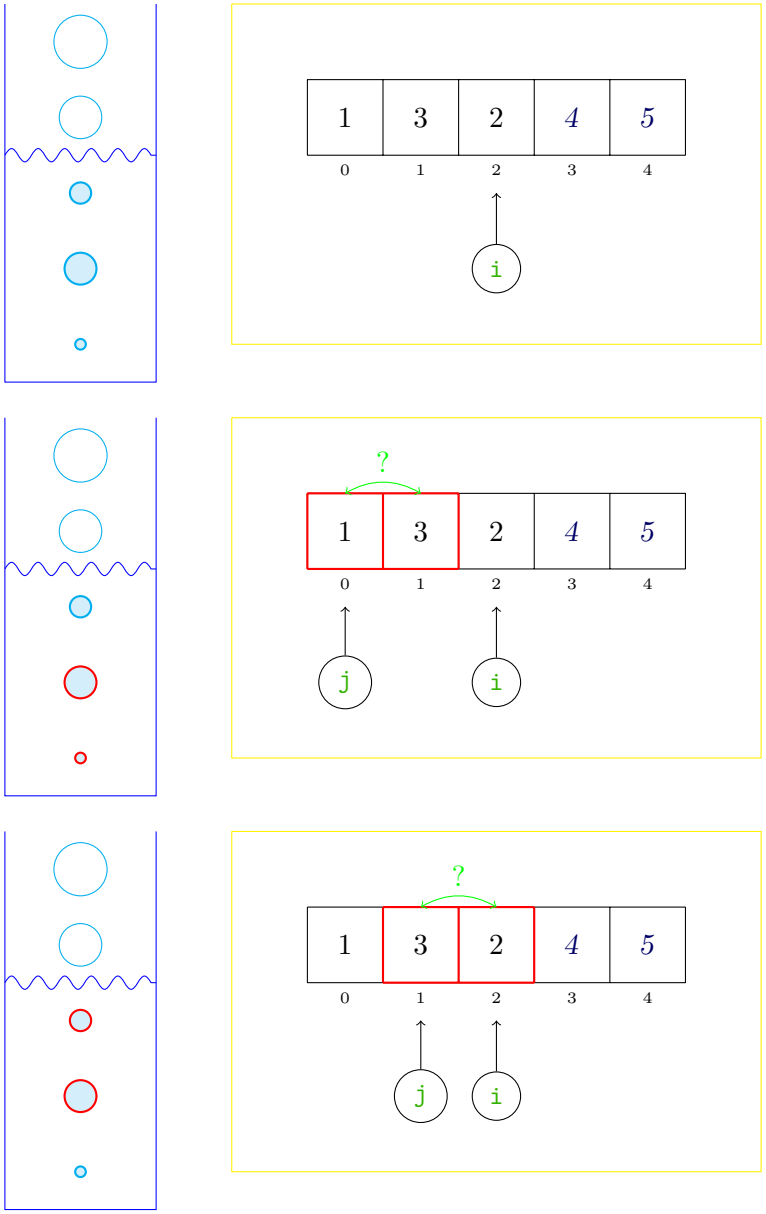
Ten algorytm również jest stabilny.



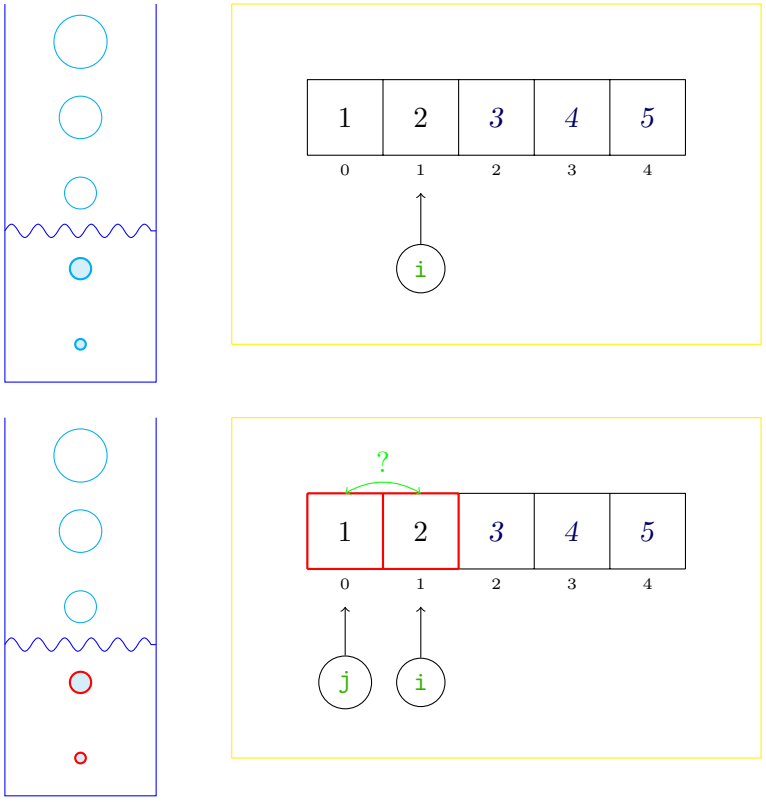
Rys. 4.13. Sortowanie bąbelkowe — przykład — krok I



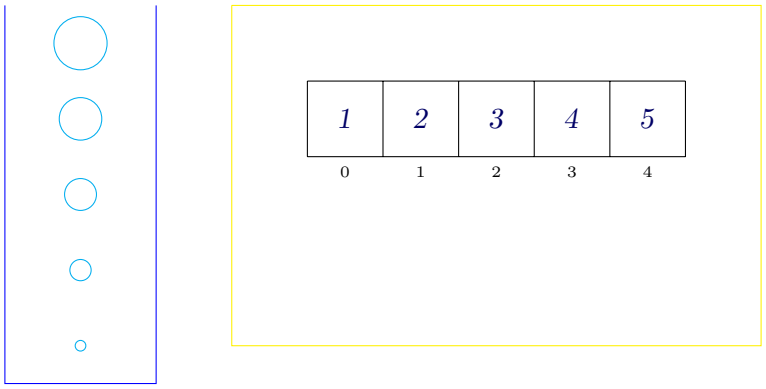
Rys. 4.14. Sortowanie bąbelkowe — przykład — krok II



Rys. 4.15. Sortowanie bąbelkowe — przykład — krok III



Rys. 4.16. Sortowanie bąbelkowe — przykład — krok IV



Rys. 4.17. Sortowanie bąbelkowe — przykład — rozwiązanie

4.2.4. Efektywność obliczeniowa

Rozważając algorytmy sortowania w sposób oczywisty obserwujemy wspomniane w pierwszym rozdziale zjawisko „jeden problem — wiele algorytmów". Rzecz jasna, trzy podane wyżej algorytmy (co zresztą zostało zapowiedziane) nie wyczerpują wszystkich możliwości rozwiązania zagadnienia sortowania tablic jednowymiarowych.

Nasuwa się więc pytanie: skoro każda z metod znajduje rozwiązanie danego problemu w poprawny sposób, jakie przesłanki powinny nami kierować przy wyborze algorytmu w zastosowaniach praktycznych?

„Leniuszki" z pewnością wybrałyby algorytm najprostszy w implementacji. Domyślamy się jednak, że nie jest to zbyt roztropne kryterium.

W tzw. *analizie algorytmów* wyróżnia się dwie najważniejsze miary efektywności obliczeniowej:

1. *złożoność czasowa* — czyli liczbę tzw. *operacji znaczących* (zależną od danego problemu; mogą to być przestawienia, porównania, operacje arytmetyczne itp.) potrzebnych do rozwiązania danego zagadnienia,

2. *złożoność pamięciowa* — czyli ilość dodatkowej pamięci potrzebnej do rozwiązania problemu.

Zauważmy, że w przypadku przedstawionych wyżej algorytmów, oprócz danej tablicy którą należy posortować i kilku zmiennych pomocniczych nie jest potrzebna żadna dodatkowa liczba komórek pamięci komputera (w przypadku bardziej złożonych metod omawianych w sem. III będzie jednak inaczej, np. konieczne będzie użycie jeszcze jednej tablicy). Skupimy się więc teraz tylko na omówieniu *złożoności czasowej*.

Przyjrzyjmy się zatem przykładowym implementacjom dwóch algorytmów dla danej tablicy `t` rozmiaru `n`. Oto kod stosujący sortowanie przez wybór:

```
// Sortowanie przez wybór:
for (int i=0; i<n-1; ++i)
{
    // znajdź indeks najmn. el. spośród t[i], ..., t[n-1];
    int j = i;
    for (int k=i+1; k<n; ++k)
        if (t[k] < t[j])          // porównanie elementów
            j = k;

    // zamiana
    int p = t[i];
    t[i] = t[j];
    t[j] = p;
}
```

A oto kod sortujący tablicę „bąbelkowo":

```
// Sortowanie bąbelkowe:
for (int i=n-1; i>0; --i)
{
    for (int j=0; j<i; ++j)
    {
        if (t[j] > t[j+1])            // porownanie parami
        {
            int p = t[j];             // zamiana
            t[j] = t[j+1];
```

```
10              t[j+1] = p;
11          }
12      }
13  }
```

Po pierwsze, musimy podkreślić, iż — formalnie rzecz biorąc — *złożoność czasową wyznacza się nie dla abstrakcyjnego algorytmu, tylko dla jego konkretnej implementacji*. Implementacja zależna jest, rzecz jasna, od stosowanego języka programowania i zdolności programisty. Nie powinno to nas dziwić, ponieważ np. za wysokim poziom abstrakcji stosowanym często w pseudokodach może kryć się naprawdę wiele skomplikowanych instrukcji języka programowania.

Po drugie, zauważmy, że *złożoność jest najczęściej zależna od danych wejściowych*. W naszym przypadku jest to nie tylko rozmiar tablicy, `n`, ale i także jej wstępne uporządkowanie. Z tego też powodu liczbę operacji znaczących powinno podawać się w postaci *funkcji rozmiaru zbioru danych wejściowych* (u nas jest to po prostu `n`) w różnych sytuacjach, najczęściej co najmniej:

1. w przypadku *pesymistycznym*, czyli dla danych, dla których implementacja algorytmu musi wykonać najwięcej operacji znaczących,
2. w przypadku *optymistycznym*, czyli gdy dane wejściowe minimalizują liczbę potrzebnych operacji znaczących.

Ciekawostka

Celowość przeprowadzenia badania przypadków pesymistycznych i optymistycznych pozostaje przeważnie poza dyskusją. Czasem także bada się tzw. złożoność czasową oczekiwaną (średnią), zakładając, że dane mają np. tzw. rozkład jednostajny (w naszym przypadku znaczyłoby to, że prawdopodobieństwo pojawienia się każdej możliwej permutacji ciągu wejściowego jest takie samo). Oczywiście wątpliwości może tutaj budzić, czy taki dobór rozkładu jest adekwatny, słowem — czy rzeczywiście modeluje to, co zdarza się w praktyce.

Nie zmienia to jednak faktu, że jest to zagadnienie bardzo ciekawe i warte rozważania. Wymaga ono jednak dość rozbudowanego aparatu matematycznego, którym niestety jeszcze nie dysponujemy.

Nietrudno zauważyć, że w przypadku przedstawionych wyżej implementacji algorytmów przypadkiem optymistycznym jest tablica już posortowana (np. $(1, 2, 3, 4, 5)$), a przypadkiem pesymistycznym — tablica posortowana odwrotnie (np. $(5, 4, 3, 2, 1)$).

Pozostaje nam już tylko wyróżnić, co możemy rozumieć w danych przykładach za operacje znaczące oraz dokonać stosowne obliczenia. Wydaje się, że najrozsądniej będzie rozważyć liczbę potrzebnych przestawień elementów oraz liczbę ich porównań — rzecz jasna, w zależności od `n`.

1. Analiza liczby porównań elementów.

1. Sortowanie przez wybór.

   Zauważamy, że liczba porównań nie jest zależna od wstępnego uporządkowania elementów tablicy. Jest ona zawsze równa $(n-1) + (n-2) + \cdots + 1 = n(n-1)/2 = n^2/2 - n/2$.

2. Sortowanie bąbelkowe. Jak wyżej, liczba porównań nie jest zależna od wstępnego uporządkowania elementów tablicy. Uzyskujemy wartość $(n-1)+(n-2)+\cdots+1 = n^2/2 - n/2$.

Obydwie implementacje algorytmów nie różnią się liczbą potrzebnych porównań. Są zatem tak samo dobre (bądź tak samo złe) pod względem tego kryterium. Zauważmy, że liczba ta jest *proporcjonalna* do n^2 (inaczej — jest rzędu n^2).

2. Analiza liczby przestawień elementów.

1. Sortowanie przez wybór.
   (a) Przypadek pesymistyczny.
   Liczba potrzebnych przestawień wynosi $n-1$.
   (b) Przypadek optymistyczny.
   Liczba potrzebnych przestawień wynosi 0.
2. Sortowanie bąbelkowe.
   (a) Przypadek pesymistyczny.
   Liczba potrzebnych przestawień wynosi $n(n-1)/2$.
   (b) Przypadek optymistyczny.
   Liczba potrzebnych przestawień wynosi 0.

Okazuje się, że w najgorszym przypadku liczba potrzebnych przestawień elementów dla naszej implementacji algorytmu sortowania przez wybór jest proporcjonalna do n, a dla sortowania bąbelkowego — jest aż rzędu n^2!

Tablica 4.1 zestawia liczbę potrzebnych przestawień elementów powyższych algorytmów i uzyskane na komputerze autora niniejszego opracowania czasy działania (w sekundach, tablice o elementach typu `int`).

Zauważamy, że użycie powyższych algorytmów już dla $n > 100000$ nie jest dobrym pomysłem (na szczęście, istnieją lepsze, naprawdę szybkie sposoby rozwiązania problemu sortowania). Ponadto, sama liczba przestawień nie tłumaczy czasu wykonania obliczeń (dlatego konieczne było także zbadanie liczby potrzebnych porównań).

Tab. 4.1. Porównanie liczby potrzebnych przestawień elementów i czasów działania dwóch algorytmów sortowania w przypadku pesymistycznym

	Liczba przestawień		Czas [s]	
n	Selection	Bubble	Selection	Bubble
100	99	4950	∼0.01	∼0.01
1000	999	499500	∼0.01	∼0.01
10000	9999	49995000	0.05	0.10
100000	99999	4999950000	6.02	10.31
1000000	999999	499999500000	631.38	1003.60

4.3. Ćwiczenia

Zadanie 4.1. Napisz funkcję, która za pomocą tylko jednej pętli `for` znajduje i wypisuje na ekran element najmniejszy i element największy danej tablicy liczb całkowitych.

Zadanie 4.2. Napisz funkcję, która za pomocą tylko jednej pętli `for` wyznaczy i zwróci trzeci największy element z danej $3 \leqslant n$-elementowej tablicy liczb całkowitych, np. dla ciągu $(3.0, 5.0, 2.0, 4.0, 1.0, 6.0)$ będzie to 4.0.

Zadanie 4.3. Niech dany będzie n-elementowy ciąg liczb rzeczywistych $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Średnią ważoną (ang. *weighted mean*) względem nieujemnego wektora wag $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ takiego, że $\sum_{i=1}^{n} w_i = 1$, nazywamy wartość $\mathsf{WM}_{\mathbf{w}}(\mathbf{x}) = \sum_{i=1}^{n} x_i w_i$. Napisz funkcję, który wyznaczy wartość WM danego ciągu względem danego nieujemnego wektora, który przed obliczeniami należy unormować tak, by jego elementy sumowały się do 1.

Zadanie 4.4. Dana jest tablica `t` składająca się z liczb całkowitych od 0 do 19. Napisz funkcję, która wyznaczy jej dominantę (modę), czyli najczęściej pojawiającą się wartość. *Wskazówka. Skorzystaj z pomocniczej, 20-elementowej tablicy, za pomocą której zliczysz, ile razy w tablicy* `t` *występuje każda możliwa wartość. Tablica powinna być zainicjowana zerami. Następnie należy znaleźć jej maksimum.*

Zadanie 4.5. Uogólnij funkcję z zad. 4.4 tak, by działała nie tylko dla tablic o elementach ze zbioru $\{0, 1, \ldots, 19\}$, ale dla dowolnego $\{a, a+1, \ldots, b\}$.

Zadanie 4.6. *Sortowanie kubełkowe.* Dana jest n-elementowa tablica `t` wypełniona liczbami naturalnymi ze zbioru $\{1, 2, \ldots, k\}$ dla pewnego k. Napisz funkcję, która za pomocą tzw. sortowania kubełkowego uporządkuje w kolejności niemalejącej elementy z `t`. W algorytmie sortowania kubełkowego korzystamy z k-elementowej tablicy pomocniczej, która służy do zliczania liczby wystąpień każdej z k wartości elementów z `t`. Na początku tablica pomocnicza jest wypełniona zerami. Należy rozpatrywać kolejno każdy element tablicy `t`, za każdym razem zwiększając o 1 wartość odpowiedniej komórki tablicy pomocniczej.

Dla przykładu, załóżmy $n = 6$, $k = 4$ oraz że dana jest tablica `t` o elementach $(4, 3, 2, 4, 4, 2)$. Tablica pomocnicza powinna posłużyć do uzyskania informacji, że w `t` jest 0 elementów równych 1, 2 elementy równe 2, 1 element równy 3 oraz 3 elementy równe 4. Na podstawie tej wiedzy można zastąpić elementy tablicy `t` kolejno wartościami $(2, 2, 3, 4, 4, 4)$, tym samym otrzymując w wyniku posortowaną tablicę.

Zadanie 4.7. Dane są dwie liczby całkowite w systemie dziesiętnym: n-cyfrowa liczba reprezentowana za pomocą tablicy `a` oraz m-cyfrowa reprezentowana za pomocą tablicy `b`, gdzie $n > m$. Każda cyfra zajmuje osobną komórkę tablicy. Cyfry zapisane są w kolejności od najmłodszej do najstarszej, tzn. element o indeksie 0 oznacza jedności, 1 – dziesiątki itd., przy czym najstarsza cyfra jest różna od 0. Napisz funkcję, która utworzy nową, dynamicznie alokowaną tablicę reprezentującą wynik odejmowania liczb `a` i `b`.

Zadanie 4.8. Dane są dwie liczby całkowite w systemie dziesiętnym reprezentowane za pomocą dwóch tablic `a` i `b` o rozmiarze $n > 1$. Każda cyfra zajmuje osobną komórkę tablicy. Cyfry zapisane są w kolejności od najmłodszej do najstarszej, tzn. element o indeksie 0 oznacza jedności, 1 – dziesiątki itd., przy czym najstarsza cyfra jest różna od 0. Napisz funkcję, która zwróci tablicę reprezentującą wynik dodawania liczb `a` i `b`.

Zadanie 4.9. Utwórz strukturę o nazwie `DzienMiesiac` zawierającą dwa pola typu całkowitego: `dzien` i `miesiac`. Następnie stwórz funkcje `poprawneDaty()`, która dla danej tablicy o elementach typu `DzienMiesiac` zwróci wartość logiczną `true` wtedy i tylko wtedy, gdy każdy jej element określa prawidłową datę w roku nieprzestępnym.

Zadanie 4.10. Zmodyfikuj funkcję z zad. 4.9 tak, by przyjmowała jako argument także rok. Sprawdzanie daty tym razem ma uwzględniać, czy dany rok jest przestępny.

Zadanie 4.11. Dany jest wielomian rzeczywisty stopnia $n > 0$ którego współczynniki przechowywane są w $(n+1$-elementowej tablicy `w` tak, że zachodzi $w(x) = \texttt{w[0]}x^0 + \texttt{w[1]}x^1 + \ldots \texttt{w[n]}x^n$, $\texttt{w[n]} \neq 0$. Napisz funkcję, która zwróci wartość $w(x)$ dla danego x.

Zadanie 4.12. Zmodyfikuj funkcję z zad. 4.11 tak, by korzystała z bardziej efektywnego obliczeniowo wzoru na wartość $w(x)$, zwanego schematem Hornera: $w(x) = (\cdots(((\texttt{w[n]}x + \texttt{w[n-1]})\,x + \texttt{w[n-2]})\,x + \texttt{w[n-3]})\cdots)\,x + \texttt{w[0]}$.

Zadanie 4.13. Niech dane będą wielomiany $w(x)$ i $v(x)$ stopnia, odpowiednio, n i m. Napisz funkcję, która wyznaczy wartości współczynników wielomianu $u(x)$ stopnia $n+m$, będącego iloczynem wielomianów w i v.

Zadanie 4.14. Dane są dwie uporządkowane niemalejąco tablice liczb całkowitych `x` i `y` rozmiarów, odpowiednio, n i m. Napisz funkcję, który zwróci uporządkowaną niemalejąco tablicę rozmiaru $n+m$ powstałą ze scalenia tablic `x` i `y`. Algorytm powinien mieć liniową (rzędu $n+m$) pesymistyczną złożoność czasową. Np. dla $\texttt{x} = (1,4,5)$ i $\texttt{y} = (0,2,3)$ powinniśmy otrzymać $(0,1,2,3,4,5)$.

Zadanie 4.15. Dana jest n-elementowa, już posortowana tablica liczb całkowitych oraz liczba całkowita x. Napisz funkcję, która za pomocą wyszukiwania binarnego (połówkowego) sprawdzi, czy wartość x jest elementem tablicy i jeśli tak, to poda pod którym indeksem tablicy się znajduje bądź zwróci -1 w przeciwnym przypadku.

Zadanie 4.16. Zaimplementuj w postaci funkcji algorytm sortowania bąbelkowego danej tablicy o n elementach typu `int`, w którym wykorzystywany jest tzw. wartownik. Wartownik to dodatkowa zmienna w, która zapamiętuje indeks tablicy, pod którym wystąpiło ostatnie przestawienie elementów w pętli wewnętrznej. Dzięki niemu wystarczy, że w kolejnej iteracji pętla wewnętrzna zatrzyma się na indeksie w. W przypadku pomyślnego ułożenia danych w tablicy wejściowej może to bardzo przyspieszyć obliczenia.

Porównaj swoją wersję algorytmu z podaną w skrypcie pod względem liczby potrzebnych operacji porównań oraz przestawień elementów w zależności od n dla tablicy już posortowanej oraz dla tablicy posortowanej w kolejności odwrotnej.

⋆⋆ **Zadanie 4.17.** Dana jest tablica o $n > 1$ elementach typu `double`. Napisz funkcję wykorzystującą tylko jedną pętlę `for`, która obliczy wariancję jej elementów. Wariancja ciągu $\mathbf{x} = (x_1, \ldots, x_n)$ dana jest wzorem $s^2(\mathbf{x}) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{\mathbf{x}})^2$, gdzie $\overline{\mathbf{x}}$ jest średnią arytmetyczną ciągu $\mathbf{x}$.

⋆ **Zadanie 4.18.** [PN i PS] Dana jest tablica o elementach typu `double`. Napisz funkcję, która sprawdza, czy każde 3 liczby z tablicy mogą być długościami boków jakichś trójkątów.

⋆ **Zadanie 4.19.** [MD] Tak zwany ciąg EKG $(a_1, a_2, \ldots)$ o elementach naturalnych jest zdefiniowany następująco: $a_1 = 1$, $a_2 = 2$ oraz a_{n+1} jest najmniejszą liczbą naturalną nie występującą wcześniej w ciągu taką, że $\mathrm{NWD}(a_n, a_{n+1}) > 1$, tzn. mającą nietrywialny wspólny dzielnik z a_n. Napisz funkcję, która wyznaczy a_n dla danego n.

⋆ **Zadanie 4.20.** [MD] Dana jest n-elementowa tablica t o elementach całkowitych, reprezentująca permutację zbioru $\{0,1,\ldots,n-1\}$. Napisz fragment kodu, który wyznaczy liczbę cykli w tej permutacji. Dla przykładu, tablica $\begin{pmatrix} 2, & 1, & 5, & 4, & 3, & 0 \\ t[0] & t[1] & t[2] & t[3] & t[4] & t[5] \end{pmatrix}$ reprezentuje permutację o 3 cyklach. Zawiera bowiem cykl o długości 3 ($t[0] = 2 \to t[2] = 5 \to t[5] = 0$), cykl o długości 2 ($t[3] = 4 \to t[4] = 3$) oraz cykl o długości 1 ($t[1] = 1$).

⋆ **Zadanie 4.21.** [MD] Ciąg Golomba $(g_1, g_2, \ldots)$ to jedyny niemalejący ciąg liczb naturalnych, w którym każda liczba i występuje dokładnie g_i razy, przy założeniu $g_1 = 1$. Napisz funkcję, która wyznaczy n-ty wyraz tego ciągu dla danego n.

i	1	2	3	4	5	6	7	8	9	10	11	12	...
g_i	1	2	2	3	3	4	4	4	5	5	5	6	...

4.4. Wskazówki i odpowiedzi do ćwiczeń

Wskazówka do zadania 4.13. Na przykład dla $w(x) = x^4 + 4x^2 - x + 2$ oraz $v(x) = x^4 + x^3 + 10$ zachodzi $u(x) = x^8 + x^7 + 4x^6 + 3x^5 + 11x^4 + 2x^3 + 40x^2 - 10x + 20$. □

Wskazówka do zadania 4.17. Należy wyprowadzić wzory (rekurencyjne) na średnią arytmetyczną i wariancję k początkowych elementów ciągu, zależne od elementu x_k oraz średniej arytmetycznej i wariancji elementów $(x_1, \ldots, x_{k-1})$. □

Wskazówka do zadania 4.18. Jeśli mamy $a \leqslant b \leqslant c$, to warunkiem koniecznym i dostatecznym tego, by dało się skonstruować trójkąt o bokach długości a, b, c, jest $c < a + b$. Uwaga. Nie trzeba sprawdzać wszystkich możliwych trójek! □

Wskazówka do zadania 4.21. Skorzystaj z pomocniczej n-elementowej tablicy liczb całkowitych. □