ROZDZIAŁ 1

Złożoność czasowa

Dokładne policzenie czasu działania programu jest bardzo pracochłonne (zależy od kompilatora, rodzaju komputera czy szybkości procesora). Dlatego nie mierzy się czasu dokładnie, lecz do pewnego rzędu wielkości. Złożoność czasowa ułatwia szacowanie czasu działania programów.

Na złożoność możemy patrzeć jak na maksymalną liczbę operacji stałych, które może wykonać program. Operacje stałe to pojedyncze operacje dodawania, mnożenia, przypisania itp. Niektórych operacji możemy nie liczyć, skupiając się na pozostałych, które wykonują się najwięcej razy. Takie operacje nazywamy *dominującymi*.

Liczba wykonywanych operacji dominujących zależy od konkretnych danych wejściowych. Zwykle chcemy wiedzieć, jak czas wykonania zależy od określonego aspektu danych. Najczęściej jest to wielkość danych, ale może być to również rozmiar kwadratowej macierzy czy wartość wczytywanej zmiennej.

1.1. Która operacja jest dominująca?

```
1 cin >> n;
2 for (int i = 1; i <= n; i++)
3     wynik++;
4 cout << wynik;
```

Operacja w wierszu 3 jest dominująca i wykona się n razy. Złożoność zapisujemy w notacji dużego O, w tym przypadku $O(n)$ — złożoność *liniowa*.

Złożoność oznacza, że program nie wykona więcej operacji niż jest to określone przez rząd wielkości. Dokładniej, w przypadku $O(n)$ program może wykonać $c \cdot n$ operacji, gdzie c jest stałą, ale nie może wykonać już na przykład n^2 operacji, gdyż jest to wyższy rząd wielkości. Innymi słowy, obliczając

złożoność, pomijamy stałe współczynniki. Jeśli więc pętla wykonywałaby się $20 \cdot n$ lub $\frac{n}{5}$ razy, to wciąż mamy złożoność $O(n)$, choć czas działania programów może być różny. Analizując złożoność, powinniśmy wyszukiwać specjalne, złośliwe przykłady danych, na których dany program będzie działał długo.

Porównanie różnych złożoności czasowych

Spróbujmy porównać podstawowe złożoności czasowe.

1.2. Złożoność stała — $O(1)$.

```
cin >> n;
cout << n * n;
```

Zawsze mamy stałą liczbę operacji.

1.3. Złożoność logarytmiczna — $O(\log n)$.

```
cin >> n;
while (n > 1) {
    n = n / 2;
    wynik++;
}
cout << wynik;
```

Wartość n jest w każdym obrocie pętli zmniejszana o połowę. Jeśli $n = 2^x$, to $\log n = x$ (zakładamy, że domyślną podstawą logarytmu jest 2). Stąd złożoność logarytmiczna.

Zastanówmy się, jak szybko będzie działał poniższy program w zależności od danych wejściowych.

1.4. Złożoność liniowa — $O(n)$.

```
cin >> n;
for (int i = 1; i <= n; i++) {
    cin >> x;
    if (x == 0)
        return 0;
}
```

Zauważmy, że jeśli pierwszą wczytaną wartością x byłoby zero, to program od razu by się zakończył. Pamiętajmy jednak że badając złożoność, szukamy

złośliwych przypadków. Ten program będzie działał najdłużej, jeśli zero nigdy nie wystąpi.

1.5. Złożoność kwadratowa — $O(n^2)$.

```
cin >> n;
for (int i = 1; i <= n; i++)
    for (int j = i; j <= n; j++)
        wynik++;
cout << wynik;
```

Wynik będzie równy $\frac{1}{2} \cdot (n^2 + n)$ (wyjaśnienie w ćwiczeniach). Obliczając złożoność, szacujemy czas działania, więc jesteśmy zainteresowani wyrażeniem, które rośnie najszybciej. Wobec tego nie tylko pomijamy stałe, ale również inne wyrażenia (w tym przypadku $\frac{1}{2} \cdot n$), które rosną wolniej. W ten sposób uzyskujemy złożoność kwadratową.

Popatrzmy na przykład, gdy złożoność zależy od kilku zmiennych.

1.6. Złożoność liniowa względem n i m — $O(n+m)$.

```
cin >> n >> m;
for (int i = 1; i <= n; i++)
    cout << i << ' ';
for (int i = 1; i <= m; i++)
    cout << i << ' ';
```

Złożoności wykładnicze

Warto wiedzieć, że istnieją inne złożoności, takie jak $O(2^n)$, $O(n!)$. Są to złożoności wykładnicze. Programy o takich złożonościach rozwiązują problemy tylko dla małych n, gdyż dla dużych działałyby bardzo długo.

Limit czasu

Obecnie komputery wykonują około 10^8 operacji w mniej niż sekundę. W typowych konkursach programistycznych, na komputerach sprawdzających rozwiązania, limity na wykonanie zadań wynoszą zazwyczaj od 1 do 10 sekund.

Podczas zawodów mamy często podane ograniczenie na wielkość danych, dzięki czemu możemy się domyślać w jakiej złożoności powinno zostać rozwiązane zadanie. Jest to zwykle duże ułatwienie, gdyż możemy szukać

rozwiązania działającego w określonej złożoności, zamiast szukać szybszych rozwiązań. Przykładowo, dla:

- $n \leqslant 1\,000\,000$ oczekiwana złożoność czasowa to $O(n)$ lub $O(n \log n)$,
- $n \leqslant 10\,000$ oczekiwana złożoność czasowa to $O(n^2)$,
- $n \leqslant 500$ oczekiwana złożoność czasowa to $O(n^3)$.

Pamiętajmy, że limity nie są ścisłe, a tylko przybliżone. Często zależą one od konkretnego zadania. Jeśli stwierdzimy, że program wykonuje dużo ponad 10^8 operacji, to powinniśmy się zastanowić, czy nie istnieje szybsze (o lepszej złożoności czasowej) rozwiązanie problemu.

Złożoność pamięciowa

Rozwiązując zadania, należy również zwrócić uwagę na limit dostępnej pamięci. Limit pamięci określa, jak dużo zmiennych możemy zadeklarować. W przybliżeniu, jeśli mamy stałą liczbę zmiennych, to złożoność pamięciowa wynosi $O(1)$. Jeżeli potrzebujemy zadeklarować tablicę z n elementami, to złożoność pamięciowa wynosi $O(n)$ (analogicznie, jak przy złożoności czasowej). W zadaniach konkursowych limit pamięci określany jest w *megabajtach*, w skrócie MB. Warto wiedzieć, że każda zmienna typu *int* zajmuje 4 *bajty*, a tablica miliona takich zmiennych zajmuje około 4 MB.

Warto zauważyć, że nie zawsze wszystkich zmiennych i struktur danych potrzebujemy używać w tym samym czasie. Dzięki temu możemy oszczędzać pamięć, używając jednej zmiennej do różnych celów (poprzez jej nadpisywanie).

Ćwiczenie

Problem: Dla danego n mamy obliczyć sumę liczb całkowitych od 1 do n.

Rozwiązanie: Zadanie możemy rozwiązać na kilka różnych sposobów. Osoba niezdająca sobie sprawy ze złożoności czasowej mogłaby zwiększać wynik o jeden.

1.7. Rozwiązanie brutalne — złożoność $O(n^2)$.

```
1 for (int i = 1; i <= n; i++)
2     for (int j = 1; j <= i; j++)
3         wynik++;
```

Zastanówmy się, jak przyspieszyć powyższy program. Zamiast pojedynczego zwiększania wyniku, moglibyśmy zwiększać go hurtowo — kolejno o $1, 2, \ldots, n$.

1.8. Rozwiązanie wolne — złożoność $O(n)$.

```
1 for (int i = 1; i <= n; i++)
2     wynik += i;
```

Co ciekawe, powyższe rozwiązanie można jeszcze bardziej przyspieszyć. Wypiszmy ciąg $1, 2, \ldots, n$, a tuż pod nim ten sam ciąg w odwrotnej kolejności. Dodajmy liczby z tych samych kolumn.

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

Po zsumowaniu, w każdej komórce pojawiła się wartość $n+1$, dzięki czemu możemy łatwo obliczyć wzór na całkowitą sumę.

1.9. Rozwiązanie szybkie — złożoność $O(1)$.

```
1 wynik = n * (n + 1) / 2;
```

W zależności od limitów w zadaniu powyższe rozwiązania mogą być akceptowane bądź nie. Przykładowo, jeśli mamy zadeklarowane, że:

- $n \leqslant 10\,000$, to wszystkie programy powinny zostać zaakceptowane,
- $n \leqslant 1\,000\,000$, to rozwiązanie brutalne wykona około 10^{12} operacji, co zwykle spowoduje przekroczenie limitu czasu. Rozwiązanie wolne i szybkie powinno zostać zaakceptowane,
- $n \leqslant 10^{10}$, to tylko rozwiązanie szybkie powinno zostać zaakceptowane.

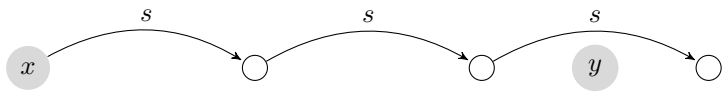
Zadania treningowe

Zadanie: Żabka ★

Konkurs: VIII Obóz Informatyczny ILOCAMP **Autor zadania:** Jacek Tomasiewicz
Pamięć: 32 MB

Mała żabka chce się przedostać na drugą stronę drogi. Początkowo znajduje się

w punkcie x, a koniec drogi jest w miejscu y. Żabka skacze zawsze o odległość równą s.



Zastanawiamy się, ile co najmniej skoków wykona żabka, nim doskoczy do końca lub przeskoczy koniec drogi.

Wejście

Pierwszy wiersz wejścia zawiera 3 liczby całkowite x, y, s ($1 \leqslant x < y, s \leqslant 10^9$) oznaczające odpowiednio: pozycję żabki, koniec drogi oraz długość skoku żabki.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą równą minimalnej liczbie skoków, po których wykonaniu żabka doskoczy do końca drogi lub go przeskoczy.

Przykład

Dla danych wejściowych:

```
10 85 30
```

Poprawną odpowiedzią jest:

```
3
```

Zadanie: Chodnik ★★

Konkurs: V Obóz Informatyczny ILOCAMP **Autor zadania:** Jacek Tomasiewicz
Pamięć: 32 MB

Edek napisał kredą na chodniku wszystkie liczby całkowite od 1 do n w losowej kolejności. Następnie poszedł do sklepu. Po powrocie zauważył, że brakuje jednej liczby. Pomóż Edkowi i powiedz, której liczby brakuje!

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($1 \leqslant n \leqslant 500\,000$), która wskazuje, ile liczb wypisał Edek.

Kolejny wiersz zawiera $n-1$ liczb całkowitych $l_0, l_1, \ldots, l_{n-2}$ ($1 \leqslant l_i \leqslant n$), gdzie l_i oznacza i-tą liczbę na chodniku (po powrocie Edka ze sklepu).

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą — liczbę, której brakuje na chodniku.

Przykład

Dla danych wejściowych:

```
5
2 3 1 5
```

Poprawną odpowiedzią jest:

```
4
```

Zadanie: Taśma ★★

Konkurs: III Obóz Informatyczny ILOCAMP **Autor zadania:** Jacek Tomasiewicz
Pamięć: 32 MB

Jaś znalazł w domu długą taśmę. Bez chwili namysłu napisał na taśmie pewien ciąg liczb całkowitych. Teraz chciałby przeciąć taśmę w pewnym miejscu, tak aby różnica między sumą liczb na jednym kawałku, a sumą liczb na drugim kawałku była jak najbliższa zeru. Chcielibyśmy znać wartość bezwzględną z tej różnicy.

Wejście

Pierwszy wiersz wejścia zawiera jedną liczbę całkowitą n ($2 \leqslant n \leqslant 1\,000\,000$), oznaczającą liczbę liczb na taśmie. Drugi wiersz wejścia zawiera n liczb całkowitych $a_0, a_1, \ldots, a_{n-1}$ ($-1\,000 \leqslant a_i \leqslant 1\,000$), gdzie a_i oznacza i-tą liczbę napisaną na taśmie.

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, równą minimalnej wartości bezwzględnej różnicy sum liczb na obu kawałkach.

Przykład

Dla danych wejściowych:

```
5
3 1 2 4 3
```

Poprawną odpowiedzią jest:

```
1
```

Rozwiązania

Przed czytaniem rozwiązań warto przeczytać punkt *Konwencje* w rozdziale *Od autora*.

Żabka

- Zadanie można rozwiązać w czasie $O(1)$.
- Różnicę między końcem drogi a punktem startowym należy podzielić przez długość skoku.

```
wynik = 1 + (y - x - 1) / s;
```

Chodnik

- Zadanie można rozwiązać w czasie $O(n)$ oraz w pamięci $O(n)$ lub $O(1)$.
- W pamięci $O(n)$ liczby można *włożyć* do odpowiadających im komórek w tablicy. Przykładowo, liczbę 5 wkładamy do komórki o indeksie 5. Na koniec wystarczy sprawdzić, której z liczb brakuje.
- W pamięci $O(1)$ można skorzystać ze wzoru na sumę liczb od 1 do n, czyli $s = \frac{n \cdot (n+1)}{2}$. Następnie od s wystarczy odjąć sumę wszystkich liczb, które pozostały na chodniku.
- Szkic rozwiązania (zakładamy, że *suma* to suma wszystkich liczb na chodniku):

```
s = (n * (n + 1)) / 2;
wynik = s - suma;
```

Taśma

- Zadanie można rozwiązać w czasie $O(n)$.
- Znając sumę wszystkich liczb oraz sumę liczb po lewej stronie, można obliczyć sumę liczb po prawej stronie.
- Szkic rozwiązania (zakładamy, że s to suma wszystkich liczb):

```
wynik = 2 * 1000 + 1;
for (int i = 0; i < n - 1; i++) {
    lewa += a[i];
    prawa = s - lewa;
    roznica = abs(lewa - prawa);
    wynik = min(wynik, roznica);
}
```
