Lekcja: Początek – wyszukiwanie binarne

"Za dużo, za mało"

W dość dawnych czasach pewna stacja radiowa organizowała dla słuchaczy zabawę: przygotowywano pewną sumę pieniędzy do wygrania, na antenie ogłaszano jej orientacyjną wielkość (na przykład "do 20 000 zł"), a następnie słuchacze dzwonili do studia. Pierwsza osoba, która podała dokładną sumę ("dwanaście tysięcy sześcset dwadzieścia trzy złote i pięćdziesiąt siedem groszy"), wygrywała ją na własność. Po każdym telefonie słuchacza rozlegał się komunikat "ZA DUŻO", jeśli podana została zbyt wielka suma, lub "ZA MAŁO", jeśli zbyt skromna.

Załóżmy, że jesteśmy jedynym dzwoniącym do studia, a suma do rozdania to całkowita liczba między 1 a 1000 złotych. Ile telefonów potrzebujemy, aby zgadnąć właściwą liczbę?

Gdybyśmy byli optymistami, zapytalibyśmy najpierw o 1000 złotych, potem o 999, potem o 998 i tak dalej... problem w tym, że przy stosunkowo rozsądnej i prawdopodobnej sumie 600 złotych wymagałoby to aż 400 telefonów. Uwzględniając koszt rachunku telefonicznego, taka strategia stałaby się całkiem nieopłacalna.

Całkiem możliwe, że trafilibyśmy już w pierwszym, lub w drugim ruchu. Ważną cechą myślenia algorytmicznego jest przygotowywanie się jednak na przypadek najgorszy, zwany zwykle pesymistycznym, choćby nawet był on mało prawdopodobny. Dla naszej strategii najgorszy przypadek nastąpi przy kwocie 1zł i zmusi nas do wykonania aż tysiąca telefonów.

Jak poradzić sobie lepiej? Zastanówmy się, co się stanie, jeśli zaczniemy od strzału w kwotę 500zł. Najprawdopodobniej nie trafimy we właściwą sumę, za to na przykład przy odpowiedzi "ZA DUŻO" wiemy już, że kwota musi mieścić się pomiędzy 1 a 499 złotych, o wszystkich większych liczbach możemy zapomnieć. Z kolei przy odpowiedzi "ZA MAŁO" właściwa kwota będzie na pewno pomiędzy 501 a 1000 złotych.

Strategię tę można powtórzyć: jeśli wynik mieści się w przedziale [500,1000], kolejnym ruchem będzie strzał w środek tego przedziału – kwotę 750 zł. Przykładowa gra (załóżmy, że nie wiedząc o tym, ścigamy nagrodę 600 zł) mogłaby przy tej strategii wyglądać tak:

Strzał	Rezultat	Wniosek
500	ZA MAŁO	Kwota w przedziale [501,1000]
750	ZA DUŻO	Kwota w przedziale [501,749]
625	ZA DUŻO	Kwota w przedziale [501,624]
562	ZA MAŁO	Kwota w przedziale [563,624]
594	ZA MAŁO	Kwota w przedziale [595,624]
609	ZA DUŻO	Kwota w przedziale [595,608]
601	ZA DUŻO	Kwota w przedziale [595,600]
598	ZA MAŁO	Kwota w przedziale [599,600]
599	ZA MAŁO	Kwota w przedziale [600,600]
600	WYGRANA!	

Do właściwej kwoty doszliśmy w dziesięciu telefonach. Przy kwocie 625 zł zgadlibyśmy już w trzeciej próbie. A czy jest możliwe, że wykonalibyśmy dużo więcej pracy? Nie jest! Popatrzmy na przedziały napisane w ostatniej kolumnie. Pierwszy z nich ma długość 500, drugi 249, kolejny 124... każdy następny jest około dwa razy krótszy od poprzedniego i jest tak bez względu na to, jakie otrzymywaliśmy odpowiedzi. Cokolwiek by się zatem nie działo, najpóźniej w dziesiątym kroku długość tego przedziału spadnie do 1, a więc będziemy znali właściwą liczbę.

Algorytm, który szuka zadanej wielkości tą metodą – dzielenia na pół przedziału, w którym może się znajdować – zwany jest **wyszukiwaniem binarnym.**

Wyszukiwanie binarne w tablicy

Wykorzystajmy wiedzę z gry, aby rozwiązać problem algorytmiczny. Mamy pewien (potencjalnie długi, ale na razie poprzestańmy na 15-elementowym) ciąg liczb, uporządkowany rosnąco:

1	4	5	9	12	18	20	21	27	32	35	48	49	51	54

Ciąg przechowujemy w tablicy *tablica[],* indeksowanej od *0* do *n-1*. Naszym zadaniem jest znaleźć pozycję (indeks tablicy), na którym jest liczba 12 – załóżmy na chwilę, że mamy pewność, iż liczba w tablicy na pewno się znajduje. Najprostszym sposobem byłoby sprawdzenie wszystkich elementów tablicy za pomocą prostej pętli. Jak zrobić to tak, aby nie napracować się za bardzo? Identycznie jak w radiowej grze: sprawdźmy najpierw, jaka liczba jest w połowie tablicy (czyli w komórce *tablica[7]*). W tym wypadku – 21, czyli za dużo, a to znaczy, że 12 na pewno znajduje się pomiędzy *tablica[0]* a *tablica[6]*. Wystarczy teraz powtórzyć operację na mniejszym, początkowym fragmencie tablicy – od indeksu *0* do *6*. W tym celu sprawdzamy środek tego fragmentu, czyli wartość *tablica[3]*...

Bardziej formalnie: staramy się znaleźć pewną liczbę *x* w tablicy *tablica[0..n-1]*, czyli podać taki indeks *i*, dla którego *tablica[i] = x*. W tym celu weźmy dwie zmienne *początek* i *koniec*, ustawmy *początek* na *0* i *koniec* na *n-1*. Będziemy starać się utrzymywać następujący fakt: element *x* znajduje się w tablicy pomiędzy indeksami *początek* i *koniec*. Patrzymy, co znajduje się dokładnie na środku pomiędzy tymi indeksami – (czyli definiujemy zmienną *środek* jako średnią *początku* i *końca*) jeśli w komórce *tablica[środek]* jest liczba większa od *x*, wiemy że trzeba szukać dalej pomiędzy *początkiem* a *środkiem*. Jeśli zaś mniejsza od *x*, szukamy między *środkiem* a *końcem*:

```
int poczatek = 0;
int koniec = n - 1;
bool znalezione = false;
do
{
        srodek = (poczatek + koniec) / 2;
        if (tablica[srodek] == x)               // trafiliśmy w x - koniec
                znalezione = true;
        else
                if (tablica[srodek] > x)        //na środku jest zbyt duży element
                        koniec = srodek - 1;    //to x w przedz. [poczatek, srodek-1]
                else                            //jeśli na środku el. jest zbyt mały
                        poczatek = srodek + 1;  //szukamy w przedz. [srodek+1, koniec]

} while (!znalezione);
```

Ten kod zadziała jednak tylko przy założeniu, że element *x* znajduje się w tablicy – po jego zakończeniu, w zmiennej *środek*. Jak zabezpieczyć się przed sytuacją, w której go nie ma? Można na przykład w ten sposób:

```
int poczatek = 0;
int koniec = n - 1;
while (poczatek < koniec)
{
    srodek = (poczatek + koniec) / 2;
    if (tablica[srodek] >= x)             //na środku jest elem. większy lub równy x...
```

```
        koniec = srodek;                    //zatem x w przedziale [poczatek, srodek]
    else                                    //na środku jest el. mniejszy od x...
        poczatek = srodek + 1;              //zatem x w przedziale [srodek+1, koniec]
}
```

Zauważmy, że w tej wersji nie sprawdzamy za każdym razem, czy nie trafiliśmy w komórkę zawierającą *x*, zamiast tego zyskując na prostocie i zwięzłości algorytmu. Procedura działa do momentu, aż *początek* i *koniec* przybiorą tę samą wartość. Jeśli element *x* jest w tablicy, musi być dokładnie w komórce *tablica[początek]*. Jeśli zaś w tym miejscu znajduje się cokolwiek innego niż *x*, wiemy, że nie było go w tablicy.

Logarytm

Ile iteracji wykona główna pętla algorytmu? Podobnie jak przy grze "za dużo, za mało", za każdym razem długość przedziału *[początek, koniec]* zmniejsza się dwukrotnie. Na początku wynosi ona n. Dla 4-elementowego przedziału wykonamy zatem dwie iteracje, dla 8-elementowego – 3 iteracje, dla 16-elementowego – 4 iteracje, i tak dalej. Widzimy (i bardzo łatwo jest pokazać), że dla $n=2^k$ wykonamy dokładnie k iteracji. Taka liczba k zwana jest **logarytmem dwójkowym** z liczby n.

Definiując precyzyjnie, logarytm dwójkowy z dowolnej liczby dodatniej a to taka liczba x, dla której $2^x=a$. Oznaczamy ją przez $\log_2 a$, przy czym często piszemy po prostu $\log a$. Dla "większości" możliwych argumentów a, nawet całkowitych, liczba $\log a$ nie jest całkowita, a nawet wymierna (na przykład $\log 12=3{,}5849\ldots$. Nam będzie na ogół przydatne jej zaokrąglenie do liczby całkowitej – w szczególności, algorytm wyszukiwania binarnego wykonuje zawsze $\lceil \log n \rceil$ (sufit z logarytmu z n) iteracji pętli.

Intuicyjnie, logarytm z liczby n mówi, ile razy możemy ją podzielić n przez 2, zanim spadnie poniżej jedności – liczbę 4 możemy podzielić dwa razy, zaś liczbę 32 – pięć razy. Jest to też "mniej więcej" liczba cyfr liczby w zapisie dwójkowym pomniejszona o jeden (na przykład $26=(11010)_2$, czyli ma pięć cyfr dwójkowych, podczas gdy logarytm z 26 to około 4,7).

Logarytm jest bardzo wolno rosnącą funkcją – logarytm z 1 000 000 (miliona) to około 20, logarytm z 10^9 (miliarda) – około 30. Dlatego algorytmy, które wykonują logarytmiczną liczbę kroków, uważane są za bardzo szybkie. Logarytm będzie pojawiał się na naszym kursie bardzo często, dlatego warto zapamiętać i dobrze zrozumieć to pojęcie.

Trochę kwestii technicznych

Co się stanie, jeśli w tablicy jest więcej niż jedno wystąpienie poszukiwanego elementu *x*? Nasza funkcja zawsze znajdzie wtedy indeks pierwszego wystąpienia. Istotnie, zauważmy, że owo pierwsze wystąpienie zawsze jest pomiędzy indeksami *początek* a *koniec*. Jeśli strzał (wskaźnik *środek*) trafi w jedną z komórek zawierających *x*, będziemy kontynuować wyszukiwanie w pierwszej połowie tablicy, być może tracąc część ostatnich wystąpień *x*, ale trzymając pierwsze wystąpienie w przeszukiwanym obszarze.

Jak zmodyfikować procedurę tak, aby znajdowała ostatnie wystąpienie *x*? Wystarczy zapewnić, aby w razie trafienia dokładnie *x* kontynuować wyszukiwanie w prawej połowie tablicy. Trzeba jednak bardzo uważać, łatwo bowiem o następujący (błędny!) kod:

```
int poczatek = 0;
int koniec = n - 1;
while (poczatek < koniec)
{
        srodek = (poczatek + koniec) / 2;
        if (tablica[srodek] <= x)
```

```
                poczatek = srodek;
        else
                koniec = srodek - 1;
}
```

Na pierwszy rzut oka wszystko jest dobrze – jeśli na środku znajduje się element *x* lub mniejszy, szukamy w przedziale *[środek, koniec]*, jeśli większy – *[początek, środek-1]*. Zobaczmy jednak, co się stanie, kiedy pozostaną nam już tylko dwa elementy do przeszukania (czyli *koniec = początek+1*), a pierwszym z nich będzie *x*. Wtedy *środek,* na skutek zaokrąglenia, ma tę samą wartość, co *początek*. Zatem instrukcja *początek = środek* nic nie zmieni, a algorytm po prostu zapętli się.

Zobaczmy, dlaczego poprzednia wersja algorytmu była wolna od tego problemu: otóż ze z względu na zaokrąglenie dzielenia przez 2 w dół, *środek* czasem jest równy *początkowi*, nigdy jednak *końcowi*. Można zatem bezpiecznie przejść do przedziału *[początek, środek],* albo *[środek+1, koniec]*, gdyż muszą one być mniejsze od przedziału *[początek, koniec]*, ale nie można bezpiecznie przechodzić do przedziału *[środek, koniec]*. Aby zadziałała druga wersja algorytmu, wystarczy zaokrąglać dzielenie do góry, na przykład w taki sposób:

```
int poczatek = 0;
int koniec = n - 1;
while (poczatek < koniec)
{
        srodek = (poczatek + koniec + 1) / 2; // dzielenie przez 2 z zaokr. w górę
        if (tablica[srodek] <= x)
                poczatek = srodek;
        else
                koniec = srodek - 1;
}
```

Teraz *środek* może być równy *końcowi*, ale nigdy nie będzie równy *początkowi*. Algorytm nie może się zatem zapętlić.