Wstęp do Informatyki
zadania ze złożoności obliczeniowej z rozwiązaniami

Przykład 1. Napisz program, który dla podanej liczby n wypisze jej rozkład na czynniki pierwsze. Oblicz asymptotyczną złożoność optymistyczną i pesymistyczną.

Rozwiązanie może wyglądać na przykład tak:

```
#include<stdio.h>

void wypisz(int n, int rozklad[], int ile_czynnikow)
{
    printf("%d = ", n);
    for(int i=0; i<ile_czynnikow; i++)
    {
        if(i > 0)
            printf(" * ");
        printf("%d", rozklad[i]);
    }
    printf("\n");
}

int main()
{
    int n, podzielone_n;
    int rozklad[1000];
    int ile_czynnikow = 0;

    scanf("%d", &n);
    podzielone_n = n;

    for(int i=2; i*i<=n; i++)
    {
        while(podzielone_n % i == 0)
        {
            podzielone_n /= i;
            rozklad[ile_czynnikow++] = i;
        }
    }
```

```
    if(podzielone_n != 1)
        rozklad[ile_czynnikow++] = podzielone_n;

    if(ile_czynnikow == 0)
        rozklad[ile_czynnikow++] = n;

    wypisz(n, rozklad, ile_czynnikow);
    return 0;
}
```

Najpierw warto uzasadnić, czemu ten program działa. Najpierw zauważyć trzeba, że każda liczba n może mieć co najmniej jeden czynnik pierwszy większy od $\sqrt{n}$ (bo gdyby były dwa, to ich iloczyn przekroczyłby n).

Celem pętli `for` jest znalezienie czynników pierwszych nie większych niż $\sqrt{n}$. Dla każdej liczby sprawdzamy, czy jest dzielnikiem i jeśli tak to traktujemy ją jako czynnik pierwszy. Nie trzeba testować pierwszości, bo na bieżąco dzielmy `podzielone_n` przez dotychczas znalezione dzielniki. Innymi słowy, na początku każdego obrotu pętli `for` wartość *podzielone_n* to n podzielone przez wszystkie swoje czynniki pierwsze mniejsze od i. Stąd od razu wynika, że jeśli i jest dzielnikiem *podzielone_n* to jest liczbą pierwszą, bo jest też **najmniejszym** (poza jedynką) dzielnikiem liczby *podzielone_n*.

Operacją dominującą jest tu policzenie warunku w pętli `while`. W funkcji wypisz wprawdzie też jest pętla, jednak wykonuje się ona tyle razy, ile jest elementów w tablicy rozklad, a wszystkie elementy (poza ewentualnie dwoma) wstawiamy do tablicy w pętli `while`.

Warunek pętla while zostanie sprawdzony co najmniej raz w każdym obrocie pętli `for`, czyli $\lfloor\sqrt{n}\rfloor$ razy plus raz na każdy znaleziony czynnik pierwszy. Każda liczba składa się z co najwyżej $\lfloor\log_2 n\rfloor$ czynników pierwszych. Więcej być nie może, bo nawet jeśli czynniki są najmniejsze możliwe (czyli równe 2) to "zmieści" się ich właśnie tyle.

Zatem algorytm działa w czasie $O(\sqrt{n}+\log n) = O(\sqrt{n})$, bo $\log n$ to wartość dużo mniejszego rzędu niż $\sqrt{n}$.

Zadanie 2. Dana jest tablica a, która zawiera parami różne liczy całkowite. W tej tablicy chcemy znaleźć drugą co do wielkości liczbę przy użyciu możliwie małej liczby porównań. Standardowe podejście dałoby około 2n operacji porównania dwóch liczb, jednak okazuje się, że można to nieco poprawić. Opis algorytmu poniżej.

Najpierw szukamy maksimum w tablicy, w dosyć specyficzny sposób:

- Porównaj $a[0]$ z $a[1]$, $a[2]$ z $a[3]$, $a[4]$ z $a[5]$ itd.
- Każde takie porównanie pozwala nam odrzucić jedną liczbę, bo na pewno nie będzie ona największa w całej tablicy. Ostatecznie wyrzucimy około połowę (w tablicy nieparzystego rozmiaru „mniejszą połowę") wszystkich liczb.
- Jeśli została jedna liczba, to znaleźliśmy maksimum. Jeśli nie - trzeba wykonać się rekurencyjnie dla liczb, które pozostały.

W ten sposób znajdziemy maksimum, nazwijmy je M. Teraz załóżmy, że zapisaliśmy sobie gdzieś na boku listę wszystkich par liczb, które porównywaliśmy,

tj. mamy listę wpisów „porównuję x z y". Łatwo zauważyć, że na pewno kiedyś musieliśmy porównać M z drugą co do wielkości liczbą w tablicy. Wystarczy więc spojrzeć na wszystkie liczby, które były kiedyś porównane z M i znaleźć największą spośród nich (to już robimy normalnie).

Zadanie polega na możliwie dokładnym policzeniu liczby wykonanych porównań. Nie jest ważny czas działania całości algorytmu (który przy dobrej implementacji wynosi $O(n)$). Można ograniczyć się do przypadku, gdy n jest potęgą dwójki.

Rozwiązanie W fazie szukania maksimum wykonamy dokładnie $n-1$ porównań, bo po każdym porównaniu odrzucamy dokładnie jednego kandydata do maksimum.

Sprawa komplikuje się nieco w drugiej fazie, więc załóżmy najpierw, że $n = 2^k$ dla pewnego całkowitego k. Zastanówmy się ile liczb było porównywanych z M w trakcie działania algorytmu. Jedna liczba była porównana, gdy było wszystkich było 2^k, następna, gdy pozostało 2^{k-1}, i tak dalej. Ostatnie porównanie było dla $2^1 = 2$ liczb. Zatem kandydatów do prawie-maksimum jest k. Spośród nich znajdziemy największą liczbę w $k-1$ porównaniach. Jeśli przez $T(n)$ oznaczymy liczbę porównań wykonaną przez algorytm, to właśnie dostaliśmy, że:

$$T(2^k) = 2^k - 1 + k - 1.$$

Z drugiej strony, dla $k+1$:

$$T(2^{k+1}) = 2^{k+1} - 1 + (k+1) - 1.$$

Czas działania pierwszej fazy potrafimy podać dokładnie, zaś czas działania drugiej wzrósł jedynie o 1.

Teraz spójrzmy, co dzieje się w ogólnym przypadku, jeśli $2^k \leq n < 2^{k+1}$, czyli $k = \lfloor \log_2 n \rfloor$. W takiej sytuacji druga faza może wymagać $k-1$ lub k porównań (zależy od n oraz od miejsca w tablicy, w którym jest maksimum). Dostaniemy więc:

$$T(n) = n + \lfloor \log_2 n \rfloor + O(1).$$

co oznacza, że wartość $n + \lfloor \log_2 n \rfloor$ to wynik z dokładnością do stałej.

Zadanie 3. Rozwiąż równanie rekurencyjne $T(n) = n + 2T(\lfloor \frac{2n}{3} \rfloor)$.

Uwaga: teraz zorientowałem się, że w pierwotnej wersji nie wstawiłem symbolu podłogi (wcześniej było $2T(\frac{2n}{3})$).

Tym razem wygodnie rozpatrzyć najpierw przypadek, gdy $n = (\frac{3}{2})^i$. Mamy wtedy:

$$T((\tfrac{3}{2})^{i+1}) = (\tfrac{3}{2})^{i+1} + 2T((\tfrac{3}{2})^i)$$

co da się zapisać w postaci:

$$a_{k+1} = (\tfrac{3}{2})^{k+1} + 2a_k.$$
$$a_0 = 1$$

Dalej są dwie metody postępowania.

Metoda 1. Postępujemy zgodnie z ogólną metodą radzenia sobie z takimi rekurencjami. Równanie jednorodne:

$$a_{k+1} = 2a_k.$$

Równanie charakterystyczne:

$$\lambda - 2 = 0.$$

Jedyny pierwiastek to $\lambda = 2$. Teraz zapisujemy „resztę" w postaci $p(k)\mu^k$:

$$(\tfrac{3}{2})^{k+1} = \tfrac{3}{2}(\tfrac{3}{2})^k$$

Z faktu z wykładu wiemy, że pewnym rozwiązaniem rekurencji jest

$$a_k = k^r q(k)\mu^k,$$

gdzie r to krotność μ jako rozwiązania równania charakterystycznego, zaś $q(k)$ to wielomian takiego samego stopnia, jak $p(k)$. W naszym przypadku $\mu = \frac{3}{2}$ nie jest rozwiązaniem, zatem $r = 0$. Wielomian $p(k)$ jest stały, więc tak samo będzie z $q(k)$ — przyjmijmy $q(k) = Q$. Wstawiamy to, co otrzymaliśmy, do równania rekurencyjnego:

$$\begin{gathered} Q(\tfrac{3}{2})^{k+1} = (\tfrac{3}{2})^{k+1} + 2Q(\tfrac{3}{2})^k \\ Q\tfrac{3}{2} = \tfrac{3}{2} + 2Q \\ -\tfrac{Q}{2} = \tfrac{3}{2} \\ Q = -3 \end{gathered}$$

Teraz wiemy, że

$$a_k = -3(\tfrac{3}{2})^k$$

to pewne rozwiązanie równania. Należy do niego dodać jakieś rozwiązanie $C2^k$ równania jednorodnego, by uzyskać równanie spełniające warunek początkowy, tzn. wartość C dobrać tak, by:

$$a_k = -3(\tfrac{3}{2})^k + C2^k$$

spełniało

$$a_0 = 1.$$

Połączmy te równania:

$$\begin{gathered} 3(\tfrac{3}{2})^0 + C2^0 = 1 \\ -3 + C = 1 \\ C = 4 \end{gathered}$$

Rozwiązaniem rekurencji jest zatem:

$$a_k = -3(\tfrac{3}{2})^k + 4 \cdot 2^k$$

Na pierwszy rzut oka trochę dziwny może wydawać się ten minus, jednak w rzeczywistości 2^k to funkcja rosnąca o wiele szybciej od $(\frac{3}{2})^k$ i wynik faktycznie jest nieujemny.

Metoda 2. Można sukcesywnie rozwijać rekurencję, tzn. za a_k wstawić $(\frac{3}{2})^k + 2a_{k-1}$, potem podstawić pod a_{k-1} itd. W ten sposób wyrazimy a_k w postaci sumy ciągu geometrycznego. Sumę tę da się zwinąć przy użyciu wzoru:

$$1 + q + q^2 + ... + q^n = afrac1 - q^n + 11 - q$$

Niezależnie od tego, z której metody korzystaliśmy, na koniec trzeba wrócić do początkowych oznaczeń. U nas $k = \log_{\frac{3}{2}} n$. Zatem

$$T(n) = T((\tfrac{3}{2})^k) = -3(\tfrac{3}{2})^{\log_{1.5} n} + 4 \cdot 2^{\log_{1.5} n}$$

$$T(n) = -3n + 4 \cdot 2^{\log_{1.5} 2 \log_2 n} = -3n + 4n^{\log_{1.5} 2}.$$

Tu skorzystaliśmy ze wzoru $\log_a c = \log_a b \log_b c$.

Ostatni krok to zapisanie wyniku dla dowolnego n. Trzeba zauważyć, że wynik dla $T((\frac{3}{2})^{k+1})$ jest nie większy niż pewna stała przemnożona przez wynik dla $T((\frac{3}{2})^k)$. Zatem w ogólności można napisać:

$T(n) = O(4n^{\log_{1.5} 2} - 3n) = O(n^{\log_{1.5} 2}) = O(n^{1.71})$

Na egzaminie nie trzeba oczywiście obliczać przybliżonej wartości logarytmu, ale wstawiłem taki wynik, żeby można było zobaczyć ile mniej więcej wychodzi.