Zadania domowe. Blok 1. Zestaw 1

Maciej Poleski

4 marca 2012

1 Tablica nieskończona

Rozwiązanie składa się z dwóch części:

1. Znalezienie górnego ograniczenia rozmiaru tablicy
2. Klasyczny binary serach w wyznaczonym przedziale

W rozwiązaniu zakładam że ∞ jest większe od każdej liczby całkowitej oraz że $\infty \geqslant \infty$.

Faza 1

```
Wejście: A - tablica zgodnie z oznaczeniami z zadania
Wyjście: m - liczba naturalna taka że 2n > m ⩾ n

extern int A[];
int m;
for(m=1 ; A[m]≠ ∞ ; m*=2);
```

Najpierw zauważmy że $A[m] = \infty$. Jest to warunek stopu pętli. Następnie $m \geqslant n$. Gdyby było inaczej to $A[m] \neq \infty$ a więc nie zaszedłby warunek stopu. W każdym kroku pętli m rośnie dwukrotnie oznacza to że jeżeli $A[m] = \infty$ to $A[\frac{m}{2}] \neq \infty$. Czyli $\frac{m}{2} < n$ więc $m < 2n$ i w końcu $2n > m \geqslant n$. Oznacza to że algorytm zwraca poprawny wynik pod warunkiem że się zakończy.
Zakończy się dlatego że funkcja wykładnicza 2^k jest rosnąca. A n jest skończone.
Na koniec zastanówmy się nad złożonością. m rośnie dwukrotnie przy każdym obiegu pętli. Początkowo $m = 1$, a na koniec $m < 2n$. Więc złożoność całego algorytmu wynosi $\Theta(\lg m)$. Funkcja logarytm binarny jest rosnąca więc $\lg m < \lg 2n$. Oznacza to że złożoność algorytmu wynosi $O(\lg 2n) = O(1 + \lg n) = O(\lg n)$

Faza 2

Dysponując obliczoną wartością m z fazy 1 natychmiast rozpoczynamy fazę 2.

```
Wejście: A - tablica zgodnie z oznaczeniami z zadania
         m - liczba uzyskana z poprzedniej fazy
         x - poszukiwana zawartość komórki
Wyjście: indeks komórki zawierającej x o ile istnieje
```

```
extern int A[];
extern int m;
extern int x;

return binary_search(A,A+m,x)-A;
```

Algorytm `binary_search` został omówiony na wykładzie. Przykładową implementację można odnaleźć w moim rozwiązaniu zadań A i B oraz co najmniej kilku zadaniach z WdP. Oczekuję zachowania takiego jak `std::lower_bound`, czyli pierwszy argument to początek przeszukiwanego przedziału, drugi to koniec przeszukiwanego przedziału, trzeci to poszukiwana wartość. Poszukiwana wartość jeżeli istnieje to jest w tym przedziale ponieważ jest liczbą całkowitą, a zgodnie z założeniem każda liczba całkowita jest mniejsza niż ∞ oraz $A[m] = \infty$ (bo $m \geqslant n$) a tablica jest posortowana niemalejąco. Wynikiem algorytmu jest pozycja, a więc po odjęciu pozycji początku przedziału uzyskujemy pozycję wewnątrz zadanego przedziału.
Złożoność algorytmu `std::lower_bound` to $O(\lg m)$. Uwzględniając fazę pierwszą złożoność obliczeniowa całego rozwiązania to $O(\lg n)$. ■

2 Cosinus

Zakładam że typ `double` jest w stanie przechować liczbę z dokładnością do k miejsca po przecinku. Nie jestem pewny na czym polega problem w zadaniu. Funkcję `sincos` ma zaimplementowana każda jednostka zmiennoprzecinkowa (FPU), jednak zakładając że mamy tą funkcję właściwie nie ma zadania (zadanie A już rozwiązałem). Dlatego dodatkowo przedstawię implementację funkcji szacującej wartość funkcji cos w oparciu o szereg Taylora.
W rozwiązaniu wykorzystuję własność funkcji $x^2 - \cos(x * \pi)$ - jest ona ściśle rosnąca i ma dokładnie jedno miejsce zerowe w zadanym przedziale.

```
Wejście: k - oczekiwana dokładność zgodnie z treścią zadania
Wyjście: x - rozwiązanie zadanego równania

extern int k;
double stopCondition = 0.1^k;

double value(double x)
{
    return x*x - cos(x*pi);
}
```

```
double solution()
{
    double l=0.0;
    double r=1.0;
    while(r-l >= stopCondition)
    {
        double c=(l+r)/2;
        (value(c)>0 ? r : l) = c;
    }
    return (l+r)/2;
}
```

Funkcja `value(x)` oblicza wartość zadanej funkcji w punkcie x. W tym celu wykorzystuje funkcję pomocniczą `cos` oraz stałą π.
Funkcja `solution` szacuje wartość rozwiązania z dokładnością do k miejsc po przecinku przy użyciu funkcji pomocniczej `value`.
Dodatkowo implementacja funkcji `cos`:

```
extern double stopCondition;

double cos(double x)
{
    double result = 1;
    double xp = 1;
    int ip = 1;
    for(int i=1 ;; ++i)
    {
        double oldResult = result;
        xp *= x*x;
        ip *= (2*i-1)*(2*i);
        result = (i%2 ? -1 : 1) * xp/ip;
        if(abs(result-oldResult) < stopCondition)
            break;
    }
    return result;
}
```

Wadą tej implementacji jest pojawianie się dużych liczb w zmiennej `ip`. Dlatego jest to tylko teoretyczny przykład metody szacowania wartości funkcji cos.
Jeżeli nie mamy do dyspozycji stałej π to możemy ją oszacować przy użyciu

tej funkcji rozwiązując równanie $\cos(2x) = 0$ (modyfikacji wymaga wtedy tylko funkcja `value`).
W praktyce na każdym komputerze funkcja `cos` jest realizowana sprzętowo i zwraca rezultat tak dokładny jak tylko można go zapisać w typie zmiennoprzecinkowym obsługiwanym na danej platformie. Podobnie ze stałą π.
Jeżeli jednak z różnych powodów chcemy uzyskać rozwiązanie znacznie bardziej precyzyjne niż umożliwia do architektura komputera, to implementując typy `int` oraz `double` jako typy o nieograniczonej precyzji (z dokładnością do k miejsc po przecinku) mamy możliwość uzyskania rozwiązania problemu przy użyciu przedstawionej implementacji. ■

3 Deski

Właściwie jest to zadanie B.

```
Wejście: n - ilość desek
         D - tablica z długościami desek
         z - minimalna akceptowana liczba fragmentów
Wyjście: największa możliwa długość fragmentu

extern int n;
extern int D[];
extern int z;

int count(int z)
{
    int r = 0;
    for(int i=0 ; i<n ; ++i)
        r += D[i]/z;
    return r;
}

int binary_search()
{
    int l = 1;
    int r = *(std::max_element(D, D+n));
    while(l < r)
    {
        int m = (l+r+1)/2;
        if(z <= count(m))
            l = m;
```

```
        else
            r = m-1;
    }
    return l;
}
```

Funkcja `binary_search` zwraca poszukiwaną wartość. Idea jest taka że funkcja `count` sprawdza ile uzyskamy fragmentów o zadanej długości. `binary_search` stara się zmaksymalizować ich długość przy jednoczesnym spełnieniu wymogu dotyczącego ich minimalnej ilości. ■