Podstawy Programowania

Wykład nr 11: Wybrane techniki implementacji algorytmów

dr hab. inż. Dariusz Dereniowski

Katedra Algorytmów i Modelowania Systemów
Wydział ETI, Politechnika Gdańska

Spamiętywanie

Spamiętywanie – technika optymalizacji rekurencji, pozwalająca na zapamiętaniu wyniku (wartości zwracanej) wywołania funkcji z danym zestawem parametrów i przywołania tej wartości przy późniejszym wywołaniu z tym samym zestawem parametrów:

- spamiętywanie może być wykorzystane tylko w przypadku funkcji obliczeniowych, w których istotna jest jedynie wartość zwracana, a nie pojawiają się efekty uboczne (takie jak: modyfikacja obszarów pamięci innych procedur, operacje wejścia/wyjścia),
- spamiętywanie może znacząco obniżyć złożoność czasową (a tym samym czas działania programu) i pamięciową (a tym samym ilości pamięci wykorzystywanej przez program) algorytmu,
- spamiętywanie może się wiązać z koniecznością implementacji dodatkowych struktur danych w celu zapamiętania wyników wcześniejszych wywołań funkcji (pewien narzut czasowy i pamięciowy).

Spamiętywanie – przykład

```
#define MAX 55

/* Funkcja obliczająca wartność n-tej liczby Fibonacciego (wersja
     rekurencyjna) */
long long fib_rek( int n ) {
    if ( n <= 2 )
        return 1;
    else
        return fig_rek( n-2) + fig_rek( n-1 );
}

/* Funkcja obliczająca wartność n-tej liczby Fibonacciego (spamiętywanie).
 Funkcja korzysta z globalnej tablicy t, w której:
  t[i] == 0 oznacza, że i-ta l. Fibonacciego nie została jeszcze obliczona
  t[i] != 0 oznacza, że i-ta l. Fibonacciego została wcześniej
                obliczona i użycie rekurencji nie jest potrzebne. */
long t[MAX];
long long fib_dyn( int n ) {
    if ( n <= 2 )
        return t[1]=t[2]=1;
    else {
        if ( t[n-2] == 0 )
            fib_dyn( n-2 );
        if ( t[n-1] == 0 )
            fib_dyn( n-1 );
        t[n] = t[n-2] + t[n-1];
        return t[n];
    }
}
```

Spamiętywanie a rekurencja – porównanie wydajności

```
 n |          wart. |      czas1 |      czas2
35 |        9227465 |       0.05 |       0.00
37 |       24157817 |       0.15 |       0.00
39 |       63245986 |       0.38 |       0.00
41 |      165580141 |       0.97 |       0.00
43 |      433494437 |       2.55 |       0.00
45 |     1134903170 |       6.61 |       0.00
47 |     2971215073 |      17.28 |       0.00
49 |     7778742049 |      45.31 |       0.00
51 |    20365011074 |     121.54 |       0.00
53 |    53316291173 |     313.72 |       0.00
```

```c
#include <stdio.h>
#include <time.h>
#include <string.h>
#define MAX 55

long long fib_rek( int n );

long t[MAX];
long long fib_dyn( int n );

int main() {
    long long f;
    clock_t start, fib_rek_czas, fib_dyn_czas;

    printf( "%2s | %15s | %10s | %10s (czas1 = czas wersji rek., czas2 = czas wersji dyn.)
        \n", "n", "wart.", "czas1", "czas2" );

    for ( int n=35; n < MAX; n+=2 ) {
        memset( (void *)t, 0, sizeof t ); // Zerujemy tablicę t do nowego testu.

        start = clock();     // Mierzymy czas działania fib_rek(n).
        f = fib_rek( n );    // Czas w sekundach będzie wynosił:
        fib_rek_czas = clock() - start;    // ((float)fib_rek_czas)/CLOCKS_PER_SEC

        start = clock();
        fib_dyn( n );
        fib_dyn_czas = clock() - start;

        printf( "%2d | %15lld | %10.2f | %10.2f\n", n, f, ((float)fib_rek_czas)/
            CLOCKS_PER_SEC, ((float)fib_dyn_czas)/CLOCKS_PER_SEC );
    }
    return 0;
}
```

Programowanie dynamiczne

Programowanie dynamiczne – technika polegająca na połączeniu spamiętywania i derekursywacji, która może być zastosowana dla problemów, w których rozwiązanie uzyskuje się poprzez rozszerzenie rozwiązania dla podproblemu (tzw. własność optymalnej podstruktury). W szczególności:

- punktem wyjścia jest zazwyczaj rozwiązanie rekurencyjne dla danego problemu,
- rekurencja charakteryzuje się m.in. tym, że rozwiązanie problemu jest sprowadzane do rozwiązania mniejszych podproblemów, z których to rozwiązań "budowane" jest rozwiązanie oryginalnego problemu,
- często się zdarza, że w toku obliczeń funkcja rekurencyjna jest wywoływana wielokrotnie dla tego samego podproblemu,
- w programowaniu dynamicznym rozwiązanie danego podproblemu wyznaczamy tylko raz, zapamiętując je w tablicy,

```
#define MAX 100
long t[MAX];
long long fib_dyn( int n ) {
    t[1] = t[2] = 1;
    for ( int i=3; i <= n; i++ )
        t[i] = t[i-2] + t[i-1];
    return t[n];
}
```

Uwaga: podany program zakłada, że funkcja nie zostanie wykonana z argumentem większym niż 99!
Ćwiczenie: napisz funkcję, która nie wykorzystuje ani tablic ani rekurencji do obliczenie zadanej liczby Fibonacciego.

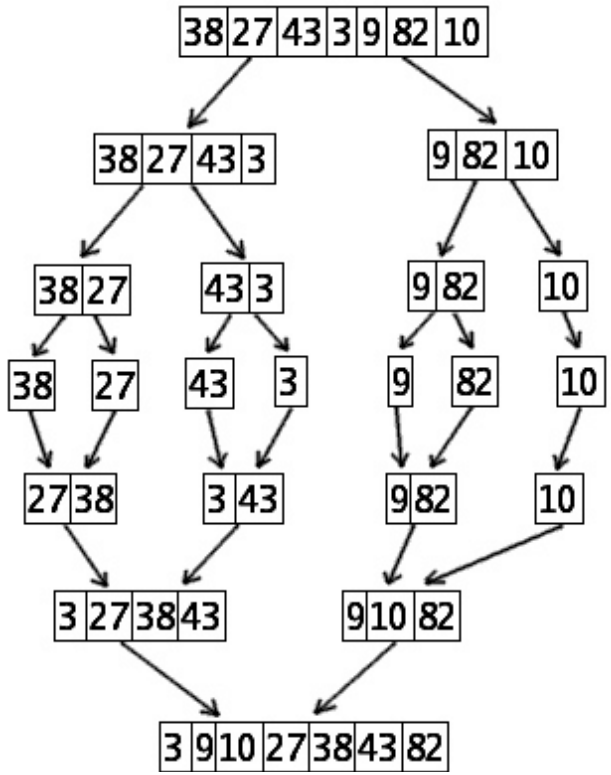
Technika “dziel i zwyciężaj”

- Algorytm typu „dziel i zwyciężaj” rekurencyjnie rozkłada problem do rozwiązania na przynajmniej dwa mniejsze podproblemy, aż do momentu, gdy podproblem nadaje się do rozwiązania wprost.
- Na każdym poziomie rekurencji, rozwiązania podproblemów są „składane”, by uzyskać rozwiązanie większego problemu.

Przykład "Dziel i zwyciężaj" – sortowanie

Przykład: sortowanie tablicy liczb przez scalanie (MergeSort)

- Podziel tablicę na dwie (prawie) równe części.
- Wywołaj rekurencyjnie procedurę MergeSort dla każdej z nich oddzielnie, chyba że pozostał już tylko jeden element.
- Scal dwa posortowane podciągi.



Przykład "Dziel i zwyciężaj" – wyszukiwanie w posortowanej tablicy

- Jeśli tablica jest posortowana, to problem wyszukiwania zadanego elementu można znacznie przyspieszyć.
- W metodzie wyszukiwania binarnego, szukany element jest porównywany ze środkowym elementem w tablicy.
- Jeśli tak się zdarzy, że ich wartości są równe, to szukany element został znaleziony.
- W przeciwnym wypadku, kontynuujemy poszukiwanie (rekurencyjnie) w odpowiedniej "połówce" tablicy.

```
typedef int element_t;

// F-cja zwraca indeks i tż t[i]==szukany, lub -1 jeśli takie i nie istnieje.
int BinarySearch( element_t szukany, element_t *t, int lewy, int prawy ) {
    if ( lewy > prawy )
        return -1; // Szukanego elementu nie ma w tablicy.
    else {
        int srodek = (lewy + prawy)/2;
        if ( t[srodek] == szukany )
            return srodek;
        else if ( t[srodek] > szukany )
            return BinarySearch( szukany, t, lewy, srodek-1 );
        else
            return BinarySearch( szukany, t, srodek+1, prawy );
    }
}
```

Przykład "Dziel i zwyciężaj" – QuickSort ("szybkie sortowanie")

Zasada sortowania tablicy t rozmiaru n metodą QuickSort:

- wybieramy dowolny element x w tablicy t,
- Tworzymy podział tablicy na dwie części: część składająca się z elementów mniejszych od x oraz z pozostałych elementów.
- Podział tworzony jest tak, że tablica t zostaje "przearanżonwana" w taki sposób, że wszystkie elementy mniejsze od x są położone na lewo od x (x może zmienić położenie w stosunku do położenia pierwotnego w t).
- Załóżmy, że po "rearanżacji" x znajduje się pod indeksem j w t, tzn. t[j]==x.
- Zauważmy, że zadanie sortowania wyjściowego ciągu n liczb t[0],...,t[n-1] zostało "zredukowane" do sortowania dwóch podciągów t[0],...,t[j-1] oraz t[j+1],...,t[n-1], każdy o długości mniejszej niż n.
- Rekurencyjne posortowanie dwóch powyższych podciągów prowadzi do tego, że tablica t będzie posortowana.

Przykładowa implementacja QuickSort

```
void zamien( int *t, int i1, int i2 ) {  // Funkcja zamienia miejscami
    int a = t[i1];                       // elementy o indeksach i1, i2
    t[i1] = t[i2];                       // w tablicy t.
    t[i2] = a;
}

/* Funkcja dokonuje podziału tablicy t, w którym element t[lewy] zostanie umieszczony pod
   indeksem granica, gdzie lewy <= granica <= prawy. Ponadto, t[i] <= t[granica] dla
   wszystkich indeksów lewy <= i < granica, oraz t[j] >= t[granica] dla wszystkich
   indeksów granica < j <= prawy.*/
int podzial( int *t, int lewy, int prawy ) {
     int x = t[lewy], granica = lewy;
     zamien( t, lewy, prawy );
     for ( int i=lewy; i < prawy; i++ )
         if ( t[i] <= x )
             zamien( t, i, granica++ );
     zamien( t, granica, prawy );
     return granica;
}

/* Funkcja sortująca 'fragment' tablicy t pod indeksami lewy,...,prawy.
   Aby posortować tablicę A należy wywołać:
      QuickSort( A, 0, (sizeof A)/(sizeof A[0])-1 );  */
void QuickSort( int *t, int lewy, int prawy ) {
    if ( lewy < prawy ) {
        int granica = podzial( t, lewy, prawy );
        QuickSort( t, lewy, granica-1 );
        QuickSort( t, granica+1, prawy );
    }
}
```

Wskaźniki do funkcji

Deklaracja wskaźnika do funkcji o argumencie typu int oraz zwracającej typ double (analogicznie dla innych typów argumentów i typów zwracanych wartości):

```
double (*wsk_fun)(int x)
```

Jeśli w programie mamy funkcję o nazwie g, której prototyp zgadza się z powyższym (czyli g przyjmuje argument typu int oraz zwraca wartość typu double), to możemy dokonać przypisania:

```
wsk_fun = g;
```

Wówczas, równoważne formy wywołania funkcji g:

```
g( 5 );
(*wsk_fun)( 5 );
wsk_fun( 5 );
```

Wskaźniki do funkcji – przykład wywołania bibliotecznego qsort

```c
#include <stdio.h>
#include <stdlib.h>

int cmp (const void * a, const void *b) {
    return  *(int*)a  -  *(int*)b;

    /* Porwnaj:
    int _va = *(int*)a;
    int _vb = *(int*)b;

    if (_va > _vb) return 1;
    if (_va < _vb) return -1;
    return 0;
    */
}

int main() {
    int t[] = { 6, 34, 21, 43, 34, 1, 0, 23, 12, 22 };

    qsort ((void*) t, (sizeof t)/(sizeof t[0]), sizeof(int), cmp);

    for (int i=0; i<(sizeof t)/(sizeof t[0]); i++)
        printf ("%d ", t[i]);
    putchar( '\n' );
    return 0;
}
```

Diagnostyka programów – warunkowe dyrektywy

```
#ifdef IDENTYFIKATOR
...
#endif
```

Fragment ... podlega kompilacji tylko wówczas jeśli identyfikator IDENTYFIKATOR jest zdefiniowany.

```
#ifndef IDENTYFIKATOR
...
#endif
```

Fragment ... podlega kompilacji tylko wówczas jeśli identyfikator IDENTYFIKATOR nie jest zdefiniowany.

- Powyższy mechanizm można użyć do umieszczania kodu, który chcemy wykonywać tylko w fazie testowania programu.
- Aby to osiągnąć, definiujemy identyfikator, na przykład #define TEST.
- Następnie, wewnątrz dyrektywy #ifdef TEST ... #endif umieszczamy dodatkowy kod używany w fazie testowania programu.
- Gdy chcemy dokonać kompilacji docelowego kodu, usuwamy tylko linię zawierającą definicję identyfikatora TEST.

Kompilacja warunkowa – przykład

```
 6 < 34
 43 > 34
 21 < 34
 6 < 21
 34 > 21
 34 = 34
 1 > 0
 12 < 22
 23 > 12
 23 > 22
 0 < 12
 1 < 12
 6 > 0
 6 > 1
 6 < 12
 21 > 12
 21 < 22
 34 > 22
 34 > 23
0 1 6 12 21 22 23 34 34 43
```

```c
#include <stdio.h>
#include <stdlib.h>

int cmp (const void * a, const void *b) {
   #ifndef TEST
      int a1 = *(int*)a, b1 = *(int*)b;
      printf( " %d %c %d\n", a1,
          (a1==b1 ? '=' : (a1>b1 ? '>' : '<') ), b1 );
   #endif
   return  *(int*)a  -  *(int*)b;
}

int main() {
   int t[] = { 6, 34, 21, 43, 34, 1, 0, 23, 12, 22 };

   qsort( (void*)t, (sizeof t)/(sizeof t[0]), sizeof(int), cmp );

   for (int i=0; i<(sizeof t)/(sizeof t[0]); i++)
      printf ("%d ", t[i]);
   putchar( '\n' );
   return 0;
}
```

Kompilacja warunkowa – przykład

```c
#include <stdio.h>
#include <stdlib.h>

// Przypisanie konkretnej wartości identyfikatorowi TEST nie jest tutaj konieczne.
#define TEST

int cmp (const void * a, const void *b) {
   #ifndef TEST
      int a1 = *(int*)a, b1 = *(int*)b;
      printf( " %d %c %d\n", a1, (a1==b1 ? '=' : (a1>b1 ? '>' : '<') ), b1 );
   #endif
   return  *(int*)a  -  *(int*)b;
}

int main() {
   int t[] = { 6, 34, 21, 43, 34, 1, 0, 23, 12, 22 };

   qsort( (void*)t, (sizeof t)/(sizeof t[0]), sizeof(int), cmp);

   for (int i=0; i<(sizeof t)/(sizeof t[0]); i++)
      printf ("%d ", t[i]);
   putchar( '\n' );
   return 0;
}
```

```
0 1 6 12 21 22 23 34 34 43
```

Uwaga: W tym przypadku instrukcje “wewnątrz” dyrektywy są usuwane z kodu przed kompilacją. Nie jest to więc równoważne użyciu instrukcji warunkowej: instrukcja warunkowa jest zawsze tłumaczona na kod wykonywalny podczas kompilacji, podczas gdy powyższa dyrektywa tylko “steruje” selekcją kodu, który będzie poddany kompilacji.