Sistemi di Calcolo (A.A. 2014-2015)
Corso di Laurea in Ingegneria Informatica e Automatica
Sapienza Università di Roma

Soluzioni esercizi riepilogativi sulla seconda parte del Modulo I – Tecniche di ottimizzazione

---

Domanda 1

Riscrivere il seguente frammento di programma C applicando manualmente constant folding:

```
double circonferenza(double raggio) {
    double PI = 3.14;
    return 2*PI*raggio;
}
```

Soluzione:

```
double circonferenza(double raggio) {
    return 6.28*raggio;
}
```

Domanda 2

Riscrivere il seguente frammento di programma applicando manualmente loop-invariant code motion:

```
void tolower(char* s) {
    int i;
    for (i=0; ; i++) {
        int c = strlen(s);
        if (i >= c) break;
        s[i] = tolower(s[i]);
    }
}
```

Soluzione:

```
void tolower(char* s) {
    int i;
    int c = strlen(s);
    for (i=0; ; i++) {
        if (i >= c) break;
        s[i] = tolower(s[i]);
    }
}
```

Domanda 3

Riscrivere il seguente frammento di programma C applicando manualmente function inlining:

```
int abs(int x, int y) { return x > y ? x-y : y-x; }

int dist(int* v, int n) {
    int i, max=0;
    for (i=1; i<n; i++)
        if (abs(v[i-1], v[i]) > max) max = abs(v[i-1], v[i]);
    return max;
}
```

Soluzione:

```
int abs(int x, int y) { return x > y ? x-y : y-x; }

int dist(int* v, int n) {
    int i, max=0;
    for (i=1; i<n; i++) {
        int a = v[i-1] > v[i] ? v[i-1]-v[i] : v[i]-v[i-1];
        if (a > max) max = a;
    }
    return max;
}
```

Nota: la definizione della funzione `abs` va lasciata perché potrebbe essere chiamata altrove.

Domanda 4

Riscrivere il seguente frammento di programma C applicando manualmente common subexpression elimination:

```
double dist(double x1, double y1, double x2, double y2) {
    return sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
}
```

Soluzione:

```
double dist(double x1, double y1, double x2, double y2) {
    double dx=x1-x2, dy=y1-y2;
    return sqrt(x*x+y*y);
}
```

Domanda 5

Riscrivere il seguente frammento di programma C applicando manualmente strength reduction:

```
int f(x) {
    return x*5;
}
```

Soluzione:

```
int f(x) {
    return x + (x << 2); // equivale a: x + x*4 = x*5
}
```

Attenzione alle precedenze tra operatori: le parentesi servono! Si noti che invece di un costoso prodotto usiamo una somma e uno shift di 2 bit a sinistra.

Domanda 6

Riscrivere il seguente frammento di programma C ottimizzando lo spazio richiesto dalla struttura:

```
struct S {
    unsigned char eta;
    int matricola;
    unsigned short data_nascita;
    char* nome;
};
```

Quanti byte occupa la struttura prima e dopo l'ottimizzazione? Assumere un sistema a 32 bit conforme con le convenzioni della System V ABI.

Soluzione:

```
struct S {
    int matricola;                  // 4 byte
    char* nome;                     // 4 byte
    unsigned short data_nascita;    // 2 byte
    unsigned char eta;              // 1 byte (+ 1 byte padding)
};
```

Prima dell'ottimizzazione la dimensione della struttura è: 1 (`eta`) + 3 (*padding*) + 4 (`matricola`) + 2 (`data_nascita`) + 2 (*padding*) + 4 (`nome`) = 16 byte.

Dopo l'ottimizzazione la dimensione della struttura è: 4 (`matricola`) + 4 (`nome`) + 2 (`data_nascita`) + 1 (`eta`) + 1 (`padding`) = 12 byte. Si noti che il padding finale serve per allineare la fine della struttura a un multiplo della size massima di ogni oggetto di tipo primitivo contenuto nella struttura (in questo caso è 4).

Domanda 7

Si consideri il seguente frammento di programma C e la sua traduzione in IA32:

```
void f() {
   int a, b, c;
   g(&a, &b, &c);
}
```

```
f: subl   $44, %esp
   leal   28(%esp), %eax
   movl   %eax, 8(%esp)
   leal   24(%esp), %eax
   movl   %eax, 4(%esp)
   leal   20(%esp), %eax
   movl   %eax, (%esp)
   call   g
   addl   $44, %esp
   ret
```

Perché il compilatore alloca 44 byte nello stack frame di `f`? I 44 byte allocati sono tutti utilizzati?

Soluzione:

Per motivi prestazionali, il compilatore fa in modo che al momento di una `call` il top della stack (`esp`) sia allineato a un indirizzo multiplo di 16. Sia x il top della stack subito prima della chiamata a `f`. Ad `f` servono 28 byte: 4 byte per l'indirizzo di ritorno, 12 byte per le tre variabili locali (di tipo `int`) e 12 byte per i tre parametri passati (di tipo `int*`). Al momento della `call g`, il top della stack deve essere ad un indirizzo $y \geq x + 4 + 24$. Il compilatore sceglie $y = x + 4 + 44 = x + 48$. In questo modo, se x è un multiplo di 16, anche y lo sarà. Dei 44 byte allocati all'ingresso di `f`, solo 24 sono effettivamente in uso. Il resto è padding. (Si noti che il compilatore allinea a un multiplo di 16 byte anche il blocco di variabili locali e c'è padding sia prima che dopo le variabili locali.)

Domanda 8

Si consideri il seguente frammento di programma C e la sua traduzione in IA32:

```
int f(int x) {
    if (0) return 2*x;
    return 3*x;
}
```

```
f: movl 4(%esp), %eax
   leal (%eax,%eax,2), %eax
   ret
```

Quali delle seguenti ottimizzazioni sono state applicate dal compilatore?

A	Dead code elimination	**B**	Function inlining
C	Constant folding	**D**	Strength reduction

Soluzione:

A. Dead code elimination. Infatti, poiché la condizione dell'`if` è sempre falsa, l'istruzione composta `if (0) return 2*x` non verrebbe mai eseguita e quindi non viene inserita nel codice IA32 generato.

Domanda 9

Si consideri il seguente frammento di programma C e la sua traduzione in IA32:

```
void f(int* v, int n, int x) {
    while (--n >= 0) v[n] = x*x;
}
```

```
f:  movl 4(%esp), %edx
    movl 12(%esp), %ecx
    movl 8(%esp), %eax
    subl $1, %eax
    js L1
    imull %ecx, %ecx
L3: movl %ecx, (%edx,%eax,4)
    subl $1, %eax
    cmpl $-1, %eax
    jne L3
L1: ret
```

Quali delle seguenti ottimizzazioni sono state applicate dal compilatore?

A	Dead code elimination	**B**	Hoisting
C	Constant folding	**D**	Strength reduction

Soluzione:

B. Hoisting. Infatti, l'istruzione `imull` che calcola `x*x` viene eseguita una sola volta prima di entrare nel ciclo `L3: … jne L3`. Si ha quindi loop-invariant code motion (hoisting).

Domanda 10

Si consideri il seguente frammento di programma C e la sua traduzione in IA32:

```
int f(int x, int y) {
    return (x-y)*(x-y);
}
```

```
f: movl 4(%esp), %eax
   subl 8(%esp), %eax
   imull %eax, %eax
   ret
```

Quali delle seguenti ottimizzazioni sono state applicate dal compilatore?

A	Dead code elimination	**B**	Function inlining
C	Common subexpression elimination	**D**	Strength reduction

Soluzione:

C. Common subexpression elimination. Infatti, l'istruzione `subl` che calcola `x-y` viene eseguita una sola volta.

Domanda 11

Si consideri il seguente frammento di programma C e la sua traduzione in IA32:

```
char sgn(int x) {
    return x > 0;
}
```

```
sgn: cmpl $0, 4(%esp)
     setg %al
     ret
```

`char f(int x, int y) {`
`  return sgn(x-y);`
`}`
	`f:  movl 4(%esp), %eax`
	`  subl 8(%esp), %eax`
	`  testl %eax, %eax`
	`  setg %al`
	`  ret`

Quali delle seguenti ottimizzazioni sono state applicate dal compilatore?

A	Dead code elimination	**B**	Function inlining
C	Common subexpression elimination	**D**	Constant folding

Soluzione:

B. Function inlining. Infatti, nel codice IA32 di `f` non c'è alcuna chiamata alla funzione `sgn`.

Domanda 12

Quale speedup ci aspettiamo per un programma se ottimizziamo di un fattore 1.5x una sua porzione che richiede il 60% del tempo di esecuzione?

Soluzione:

Applicando la legge di Amdahl con $k = 1.5$ e $\alpha = 0.6$ si ottiene $S = \frac{1}{\frac{0.6}{1.5}+1-0.6} = 1.25x$.

Domanda 13

Qual è lo speedup massimo ottenibile per un programma se ottimizziamo una sua porzione che richiede il 30% del tempo di esecuzione?

Soluzione:

Applicando la legge di Amdahl con $k \to \infty$ e $\alpha = 0.3$ si ottiene $S_{max} = \lim_{k\to\infty} \frac{1}{\frac{0.3}{k}+1-0.3} = \frac{1}{0.7} < 1.43x$.

Domanda 14

Conviene rendere 2 volte più veloce una funzione che richiede il 10% del tempo di esecuzione, oppure velocizzare del 10%[1] una funzione che richiede il 90% del tempo di esecuzione?

Soluzione:

Applicando la legge di Amdahl con $k = 2$ e $\alpha = 0.1$ si ottiene $S_{2x,10\%} = \frac{1}{\frac{0.1}{2}+1-0.1} = 1.05x$.

Se invece $k = 1.11$ (più veloce del 10%) e $\alpha = 0.9$ si ottiene $S_{1.11x,90\%} = \frac{1}{\frac{0.9}{1.11}+1-0.9} = 1.09x$.

Conviene quindi (di pochissimo) velocizzare del 10% una funzione che richiede il 90% del tempo di esecuzione.

---

[1] Velocizzare una funzione A del 10% significa che $T'_A = T_A \cdot 0.9$, quindi uno speedup per A pari a $\frac{T_A}{T'_A} = \frac{1}{0.9} = 1.11x$.