Optimización respecto a las estucturas de datos

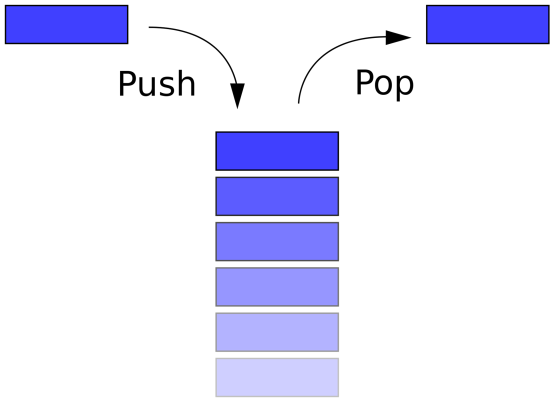
El algoritmo debe realizar las siguientes tareas:

- generar, almacenar y mostrar por pantalla una secuencia
- capturar y almacenar la secuenca ingresada por el jugador
- Comparar las dos secuencias y establecer el puntaje

Optimización respecto a las estucturas de datos

PRIMERA IDEA: Usar una estructura pila (LIFO)

Veamos una implementación con lista enlazada en C




```c
#include <stdio.h>

typedef int item;
const item indefinido=9999;

struct nodo{
      item dato;
            nodo *sig;
       };

typedef nodo* Pila;

Pila pilaVacia();
int esPilaVacia(Pila l);
Pila push(Pila l, char x);
Pila pop(Pila l);
item top(Pila l);
void escribir(Pila l);
```

Optimización respecto a las estucturas de datos

Cuál es el costo de cada operación?

```
1  int esPilaVacia(Pila l){
2    return l==NULL;}
3
4  Pila push(Pila l, item x){
5    nodo *aux;
6    aux=new nodo;
7    aux->dato=x;
8    aux->sig=l;
9          l=aux;
10         return l;}
11
12 item top(Pila l){
13   if(!esPilaVacia(l))
14     return l->dato;
15   else
16     return indefinido;}
```

```
1
2  Pila pop(Pila l){
3    if(!esPilaVacia(l)){
4      nodo * aux=l;
5      l=aux->sig;
6      delete aux;
7    }
8    return l;}
9
10 void escribir(Pila l){
11   printf("\n PILA ->");
12   while(!esPilaVacia(l)){
13     printf(" %d ->",l->dato);
14                   l=l->sig;}
15   printf(" FIN\n");
16 }
```

Optimización respecto a las estucturas de datos

Volviendo al problema:ALGORITMO (pseudocódigo)

```
WHILE not EndOfSequence
    item=aleat(N)
    p1=push(p,item)
    display(item)

END

WHILE not EndOfSequence
    item=GetItem()
    p2=push(p2,item)
    display(item)

END
```

Ahora tendríamos que comparar p1 y p2
Cuál sería el costo?

Optimización respecto a las estucturas de datos

PRIMERA IDEA: Usar una estructura pila (LIFO)

Caso1:
pc: [1,**2**]
player:[] (pila vacia)
player: player +0 ptos
Caso2:
pc: [1,**2**]
player: [1,**2**]
Listas equivalentes

Caso 3:
pc: [1,2,**2**]
player: [1,2,**3**]
Caso 4:
pc: [1,2,3,**4**]
player: [1,2,**3**]
Player +3 ptos

Se puede mejorar?
Era esta la estructura de datos mas acertada para el problema?

Optimización respecto a las estucturas de datos

PENSEMOS OTRA SOLUCION: Usar una estructura lista (FIFO)
Como se realizaria entonces la comparación?
Recordemos

Optimización respecto a las estucturas de datos

Entonces

Caso1:
pc: [**1**,2]
player:[] (fila vacia)
player: player +0 ptos
Caso2:
pc: [**1**,2]
player: [**1**,2]
Listas equivalentes

Caso 3:
pc: [**1**,2,2]
player: [**1**,2,3]
Caso 4:
pc: [**1**,2,3,4]
player: [**1**,2,3]
Player +3 ptos

Cuanto mejora?

Optimización respecto al hardware

Cómo hacer más rápido nuestros cálculos?

Optimización respecto al hardware

Cómo hacer más rápido nuestros cálculos?

- Escribir y leer más rápido (I/O ¿ veloc)
- Dar vueltas más rápido la manija (¿ veloc clock)
- Mejorar la tecnología (mejor CPU)
- Limitar el acceso a memoria no-uniforme y multi-core
- Arquitecturas multi-cores. Paralelización
- Compiladores que permiten optimización para aprovechar la CPU (pipelining, unidades vectoriales y superescalares , etc)

WTPC workshop en técnicas de programación científica

Optimización respecto al hardware

El tipo de operaciones importa!!!

- + - *
- / % sqrt()
- Func. trascendentales
- Pow(x,y) x,y reales

- Op. más veloces,0.5x-1x
- Op. velocidad media,5x-10x
- Op. lentas, 20x-50x
- Op. costosas, muy lentas,+100x

Pipelining operaciones veloces.
Usar BLAS-LAPACK (álgebra lineal, son + y *)

Técnicas de optimización (compilador)

Opt. Escalar

- Copy propagation
- Const folding
- Strength reduction
- Eliminación subexpresiones comunes
- Renombrado vbles

Opt. Lazos

- Loop invariant
- Loop unrolling
- Intercambio orden loop
- Fusion/fision loop

Inlining

- Reemplaza una porción de código por otro equivalente + veloz)

WTPC workshop en técnicas de programación científica

Copy propagation

Antes:
x=y
z=1+**x**

–

Después:
x=y
z=1+Y

El compilador elimina la dependencia
(en caso de ser posible)

Const folding

Antes
a=100;
b=200;
sum=a+b;

–
Después:
sum=300;

El compilador pre-calcula el resultado una única vez en tpo de compilación. Elimina código redundante.

Strength Reduction

Antes
x=pow(y,2.0);
a=b/2.0;

Después:
x=y*y;
a=b*0.5;

Pow, / son las operaciones más costosas

Si el compilador puede saber que se están realizando la potencia de un num entero pequeño, o el denominador de una división es una constante; entonces puede reemplazar estas operaciones por productos que es menos costoso

Eliminación subexpresiones comunes

Antes
d=c*(a/b);
e=(a/b)*2.0;

Después:
x=a/b;
d=c*x;
e=x*2.0;

La idea es eliminar el cálculo doble de la /

Solo si la subexpresion es un cálculo costoso o si resulta en la reducción de registros a utilizar

Renombrado de variables

Antes

x=y*z;

q=r+**x***2;

x=a+b;

–

Después:

x0=y*z;

q=r+x0*2;

x=a+b;

El código original tiene una dependencia de salida, el segundo no (x sigue teniendo el mismo resultado final)

Loop invariante

Antes

```
DO i=1,n
  a(i)=b(i)+c*d
  e=g(n)
END DO
```

Después:

```
tempx=c*d;
Do i=1,n
  a(i)=b(i)+temp
END DO
e=g(n)
```

Loop invariant= código que no cambia dentro de un lazo.

No es necesario que se ejecute iterativamente

Loop unrolling

Antes

```
DO i=1,n
 a(i)=a(i)+b(i)
END DO
```

Después:

```
Do i=1,n,4
 a(i)=a(i)+b(i)
 a(i+1)=a(i+1)+b(i+1)
 a(i+2)=a(i+2)+b(i+2)
 a(i+3)=a(i+3)+b(i+3)
ENDO
```

X el compilador

Intercambio orden loop

Antes

```
DO i=1,ni
  DO j=1,nj
    a(i,j)=b(i,j)
  END DO
END DO
```

Después:

```
DO j=1,nj
  DO i=1,ni
    a(i,j)=b(i,j)
  END DO
END DO
```

Depende : en Fortran (después), en C (antes)

Loop fisión/fusión

Fisionar

```
DO i=1,n
  a(i)=b(i)+1
END DO
DO i=1,n
  c(i)=a(i)/2
END DO
DO i=1,n
  d(i)=1/c(i)
END DO
```

Fusionar

```
DO i=1,n
 a(i)=b(i)+1
 c(i)=a(i)/2
 d(i)=1/c(i)
END DO
```

Innlining

Antes

```
DO i=1,n
  a(i)=f(i)
END DO
...

REAL FUNCTION f(x)
  f=x*3
END FUNCTION f
```

Después:

```
DO i=1,n
  a(i)=i*3
END DO
```

Elimina overhead del llamado a la función.

En gral el compilador tiene esta funcionalidad para altos niveles de optimización

Optimizaciones del compilador

https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc/Optimize-Options.html

`-O2` turns on all optimization flags specified by `-O`. It also turns on the following optimization flags:

```
-fthread-jumps
-falign-functions  -falign-jumps
-falign-loops  -falign-labels
-fcaller-saves
-fcrossjumping
-fcse-follow-jumps  -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize -fdevirtualize-speculatively
-fexpensive-optimizations
-fgcse  -fgcse-lm
-fhoist-adjacent-loads
-finline-small-functions
-findirect-inlining
-fipa-sra
-fisolate-erroneous-paths-dereference
-foptimize-sibling-calls
-fpartial-inlining
-fpeephole2
-freorder-blocks  -freorder-functions
-frerun-cse-after-loop
-fsched-interblock  -fsched-spec
-fschedule-insns  -fschedule-insns2
-fstrict-aliasing -fstrict-overflow
-ftree-switch-conversion -ftree-tail-merge
-ftree-pre
-ftree-vrp
```

Optimización

M. Graciela Molina

m.graciela.molina@gmail.com