Diseño de Algoritmos - Algoritmos II

Nazareno Aguirre, Sonia Permigiani, Gastón Scilingo, Simón Gutiérrez
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Clase 6: Memoization (Funciones de Memoria)

Estrategias de Diseño de Algoritmos

Las estrategias de diseño de algoritmos nos brindan herramientas para atacar el problema de construir soluciones algorítmicas para problemas. Hemos visto ya algunas de estas estrategias, principalmente Fuerza Bruta y Divide & Conquer/Decrease & Conquer.

Vimos también una importante estrategia de optimización denominada Programación Dinámica, basada en el almacenamiento del cálculo de las soluciones a subproblemas, y la inversión en el orden en que éstos se resuelven, respecto de una solución algorítmica recursiva inicial.

Veremos ahora otra forma de hacer Programación Dinámica, denominada memoization.

Memoization

Memoization es una técnica aplicable en las mismas circunstancias que Programación Dinámica, es decir, cuando tenemos un problema P, para el cual:

- ya se cuenta con una solución recursiva (top-down, en el sentido que para resolver el problema lo divide en subproblemas más pequeños, y combina sus soluciones para resolver el problema original), y
- los subproblemas a resolver NO son independientes, sino que comparten a su vez subproblemas (es decir, se "solapan"), y por lo tanto requieren recomputar varias veces las soluciones a algunos subproblemas comunes.

Resolver el problema mediante Memoization consiste en evitar recalcular resultados de llamadas a funciones previas, manteniendo una cache de valores previamente computados (cache de soluciones a subproblemas anteriores).

Nuevamente, Memoization cambia eficiencia temporal por espacio de almacenamiento, pues debe mantener una cache (mayor espacio), pero evita recomputar soluciones a subproblemas (menor tiempo).

Memoization: Esquema General

Memoization puede describirse, brevemente, mediante el siguiente esquema:

programa a optimizar.

```
T program(T1 x1, T2 x2, ..., Tk xk) {
   ...
}
```

cambiamos llamadas a program por llamadas a memoProgram

estructura para almacenar resultados

```
Map<(T1 x1, T2 x2, ..., Tk xk), Tk> cache;
```

```
T memoProgram(T1 x1, T2 x2, ..., Tk xk) {
    if (!cache.containsKey(x1, ..., xk) ) {
        cache.put((x1, ..., xk), program(res));
    }
    return cache.get((x1, ..., xk));
}
```

sólo llamamos a program si no fue previamente invocado con los parámetros actuales

Ejemplo: Números de Fibonacci

Consideremos nuevamente el problema de computar el n-ésimo número de Fibonacci:

```
public static int basicFibonacci(int i) {
    if (i<0) throw new IllegalArgumentException("undefined for nonnegative integers");
    else {
        if (i<=1) return 1;
        else return (basicFibonacci(i-1)+basicFibonacci(i-2));
    }
}
```

Sabemos que esta función es muy ineficiente (complejidad exponencial).

Ejemplo: Números de Fibonacci

Aplicar Memoization sobre la función anterior da como resultado lo siguiente:

```
private static Map<Integer,Integer> cache = new HashMap<Integer,Integer>();

public static int memoizedBasicFibonacci(int i) {
    if (i<0) throw new IllegalArgumentException("Fibonacci only defined for nonnegative integers");
    else {
        if (i<=1) return 1;
        else return (memoFibonacci(i-1)+memoFibonacci(i-2));
    }
}

public static int memoFibonacci(int i) {
    if (!cache.containsKey(i)) {
        cache.put(i, memoizedBasicFibonacci(i));
    }
    return cache.get(i);
}
```


Ejemplo: Números de Fibonacci

La función anterior se puede simplificar incluso más, eliminando la recursión mutua:

```
private static Map<Integer,Integer> cache = new HashMap<Integer,Integer>();

public static int memoFibonacci(int i) {
   if (i<0) throw new IllegalArgumentException("undefined for nonnegative integers");
   else {
      if (!cache.containsKey(i)) {
         if (i<=1) {
            cache.put(i, 1);
         }
         else {
            cache.put(i, memoFibonacci(i-1)+memoFibonacci(i-2));
         }
      }
      return cache.get(i);
   }
}
```


Memoization y Programación Funcional

Implementar Memoization en programación funcional es ligeramente más complejo, dado que no podemos llevar la cache como "memoria global", sino que la tenemos que "arrastrar" como parámetro de la función memoizada

```
fibo :: Int -> Int
fibo x = lookUp x (memoFibo x [])

memoFibo :: Int -> [(Int, Int)] -> [(Int,Int)]
memoFibo x cache | defined cache x = cache
                 | x<0             = error "undefined"
                 | x<=1            = store cache x 1
                 | otherwise       = store cacheSnd x (y1+y2)
                 where cacheFst = memoFibo (x-1) cache
                       cacheSnd = memoFibo (x-2) cacheFst
                       y1 = lookUp (x-1) cacheSnd
                       y2 = lookUp (x-2) cacheSnd

lup :: Int -> [(Int, Int)] -> Int
lup x [] = error "undefined"
lup x1 ((x2,y):ps) | x1==x2 = y
                   | x1/=x2 = lookUp x1 ps

defined :: [(Int,Int)] -> Int -> Bool
defined ps x = elem x (map fst ps)

store :: [(Int, Int)] -> Int -> Int -> [(Int,Int)]
store ps x y | defined ps x = ps
             | otherwise    = (x,y):ps
```

Algunos Puntos Importantes sobre Memoization

- Es una forma de "implementar" Programación Dinámica.
  - Se aplica en los mismos casos en que se aplica programación dinámica (se debe contar ya con una solución recursiva a un problema, con "solapamiento")
- Sólo es correcto usar memoization si el programa a memoizar dar exactamente los mismos resultados cuando se invoca con los mismos parámetros
  - no tiene efectos colaterales
  - no depende de estado mutable (fuera de los parámetros)

Memoization y Lenguajes Interpretados

Además de poder utilizar memoization como técnica de implementación para algunas rutinas particulares, en los lenguajes interpretados se puede cambiar el evaluador de programas para que aplique memoization sobre algunas rutinas

Haskell: memoize package

Python: repoze.lru

```python
from repoze.lru import lru_cache

@lru_cache(maxsize=500)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Ruby: memoize gem

```ruby
# Inefficient fibonacci method
def fib(n)
  return n if n < 2
  fib(n-1) + fib(n-2)
end

fib(100) # Slow

memoize(:fib)
fib(100) # Fast

# Or store the cache to a file for later use
memoize(:fib, "fib.cache")
fib(100) # Fast
```


Resumen de la Clase

- La Programación Dinámica es una técnica de diseño de algoritmos fuertemente ligada a recursión, que intenta mejorar la eficiencia de tales soluciones invirtiendo el orden de cómputo de soluciones a subproblemas.
- Memoization es una forma de implementar soluciones basadas en programación dinámica, que se basa en mantener una cache de resultados a invocaciones del programa a optimizar.
- Memoization puede aplicarse de forma ad hoc a programas que tengan las mismas características que aquellos sobre los que se aplica Programación Dinámica, i.e., que resuelvan un problema mediante soluciones a subproblemas "con solapamientos".
- En el caso de los lenguajes interpretados, existen librerías que permiten aplicar memoization alterando la evaluación de programas, sin necesidad de "implementarla" manualmente en el código del programa a optimizar.