Tarea de programación dinámica

Dalia Camacho, Gabriela Vargas, Elizabeth Monroy

November 18, 2018

Partición

El problema de partición tiene la siguiente formula

$$Particion(m,n) = \begin{cases} 1, & si \quad m=1 \quad o \quad n=1 \\ Particion(m,m) & si \quad m<n \\ Particion(m,n-1) & si \quad m=n \\ Particion(m,n-1)+Particion(m-n,m) & si \quad m>n \end{cases} \tag{1}$$

El código de partición recursivo es el siguiente:

```
ParticionRec <- function(m,n){
  if(m==1 || n==1){
    return(1)
  }
  if(m<n){
    return(ParticionRec(m,m))
  }else if(m==n){
    return(1+ ParticionRec(m,n-1))
  }else{
    return(ParticionRec(m,n-1) + ParticionRec(m-n,n))
  }
}
```

Como ejemplo corremos el código partición para $m=4$ y $n=4$, lo hacemos para valores pequeños, ya que para valores grandes la pila se llena y el programa truena.

```
ParticionRec(4,4)
```

```
## [1] 5
```

El código recursivo se puede reescribir de forma iterativa. Para esto se define una matriz $M \in \mathbb{R}^{m\times n}$ si $n<m$, o bien una matriz $M \in \mathbb{R}^{m\times m}$ si $n \geq m$, dado que $Particion(m,n) = Particion(m,m)$ sin $n>m$. Dentro de la matriz M se van guardando los valores de la función *Partición* por lo que $M_{i,j} = Particion(i,j)$. M se inicializa con unos en la primera fila y en la primera columna. A partir de eso se define un ciclo sobre las filas empezando en la segunda fila y dentro de este un ciclo sobre las columnas, el cual también empieza en dos. Dentro de los ciclos se va actualizando la matriz M de acuerdo al algoritmo de partición.

```
ParticionIter <- function(m,n){
  if(m==1 || n==1){
    return(1)
  }
  if(m<n){
    n <- m
  }
  M <- matrix(1, nrow = m, ncol = n)
  for(i in 2:m){
    for (j in 2:n) {
```

```
      if(i<j){
        M[i,j] <- M[i,i]
      }else if(i==j){
        M[i,j] <- 1 + M[i,j-1]
      }else{
        M[i,j] <- M[i,j-1] + M[i-j,j]
      }
    }
  }
  return(M[i,j])
}
```

Comprobamos que el programa iterativo regrese los mismos valores para $m = 4$ y $n = 4$.

```
ParticionIter(4,4)
```

```
## [1] 5
```

Ahora lo probamos para $m = 100$, $n = 100$

```
ParticionIter(100,100)
```

```
## [1] 190569292
```

Ackerman

El algoritmo de Ackerman esta dado de la siguiente manera:

$$A(m,n) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & m > 0 \quad y \quad n = 0, \\ A(m-1, A(m,n-1)) & m > 0 \quad y \quad n > 0 \end{cases} \tag{2}$$

El código recursivo está dadao por:

```
AckermanRec <- function(m,n){
  if(m==0){
    return(n+1)
  }else if(m>0 & n==0){
    return(AckermanRec(m-1,1))
  }else if(m>0 & n>0){
    return(AckermanRec(m-1, AckermanRec(m,n-1)))
  }
}
```

Vemos el resultado para $m = 3$, $n = 3$.

```
AckermanRec(3,3)
```

```
## [1] 61
```

Este algoritmo no lo pudimos reescribir en términos de programación dínamica o de forma iterativa, dada la forma en que crece el algoritmo no es posible predeterminar un arreglo del tamaño necesario para guardar los valores. Anteriormente vimos que en el caso de $m = 3$ y $n = 3$ Ackerman da como resultado 61, esto continúa aumentando conforme aumenta m. Seguimos el algoritmo para valores de $m =\in \{0, 1, 2, 3\}$ e intentamos encontrar una fórmula cerrada cada uno de esos casos. El proceso que seguimos fue parecido a un proceso iterativo. Calculamos primero la fórmula para $m = 0$ después para $m = 1$ y así sucesivamente. Además para cada m la n la recorrimos empezando en cero. De esta forma ya conocíamos la fórmula para $A(m-1,n)$ para cualquiern n y el valor de $A(m,n-1)$. Con esto era posible calcular el siguiente valor sin utilizar la cola. Para el caso con $m=0$ la solución se sigue de la definición del algoritmo.

$$A(0,n) = n+1.$$

Para $m=1$ $A(1,0) = A(0,1) = 2$. $A(1,n) = A(0,A(1,n-1)) = A(0,A(0,A(1,n-2)))$
$= A(0,A(0,A(0,A(0\ldots A(0,A(1,0)))))$
$= 1+1+\cdots+1+2 = n+2$.

Con lo que

$$A(1,n) = 2.$$

Para $m=2$ $A(2,0) = A(1,1) = 3$

$A(2,1) = A(1,A(2,0)) = 2+3 = 5$

$A(2,2) = A(1,A(2,1)) = 7$

$A(2,3) = A(1,A(2,2)) = 9$

A partir de esto se tiene que:

$$A(2,m) = 2(n+1)+1$$

Finalmente lo hacemos para $m=3$ $A(3,0) = A(2,1) = 5$

$A(3,1) = A(2,A(3,0)) = 13$

$A(3,2) = A(2,A(3,1)) = A(2,13) = 29$

$A(3,3) = A(2,A(3,2)) = 61$

$A(3,4) = A(2,A(3,3)) = 125$

Con esto podemos ver que $A(3,n) = 8(2^n-1)+5$

Calculamos algunos valores para $m=4$

$A(4,0) = A(3,1) = 13$

$A(4,1) = A(3,A(4,0)) = A(3,13) = 8(2^{(13)}-1)+5 = 65533$

$A(4,2) = A(3,A(4,1)) = A(3,65533) = 8(2^{(65533)}-1)+5$

Intentamos evaluar $A(4,2)$ con la fórmula anterior y nos da infinito:

```
8*(2^(65533)-1)+5
```

```
## [1] Inf
```

Por lo que nos detenemos en este punto, ya que el algoritmo se vuelve poco manejable.

Construimos una función de Ackerman para valores de $m \in 0,1,2,3$

$$A_restringido(m,n) = \begin{cases} n+1, & m=0 \\ n+2, & m=1 \\ 2(n+1)+1, & m=2 \\ 8(2^(n)-1)+5, & m=3 \end{cases} \tag{3}$$

```
AckermanRestringido <- function(m,n){
  if(m==0){
    return(n+1)
  }else if(m==1){
```

```r
    return(n+2)
  }else if(m==2){
    return(2*(n+1)+1)
  }else if (m==3){
    return(8*(2^(n)-1)+5)
  }else{
    stop("Value of m>3, problem grows uncontrollably")
  }
}
```

Ahora lo probamos

```r
AckermanRestringido(0,2)
```

```
## [1] 3
```

```r
AckermanRestringido(0,189)
```

```
## [1] 190
```

```r
AckermanRestringido(1,0)
```

```
## [1] 2
```

```r
AckermanRestringido(1,189)
```

```
## [1] 191
```

```r
AckermanRestringido(2,0)
```

```
## [1] 3
```

```r
AckermanRestringido(2,189)
```

```
## [1] 381
```

```r
AckermanRestringido(3,0)
```

```
## [1] 5
```

```r
AckermanRestringido(3,189)
```

```
## [1] 6.277102e+57
```

Ruta óptima

Pregunta 3

A continuación, se presenta la implementación de un algoritmo Bottom-Up en R para encontrar la distancia más corta entre dos nodos de un grafo de 10 vértices.

```
dijkstra<-function(s,w){
  n<-ncol(w) #Número de nodos
  #arreglos donde se guardarán los datos
  dist <- numeric(n)
  visited <- numeric(n)
  path <- numeric(n)

  #Se inicializa arreglo de distancias desde nodo de inicio a los otros
  #nodos en el grafo
  for(i in 1:n){
    path[i] <- -1
    dist[i] <- w[s,i]
  }

  #contador que lleva el registro de los nodos recorridos
  count<-2

  while(count <= n){
    min<-Inf

    for(j in 1:n){
      #Se identifica la mínima distancia en el arreglo dist a un nodo que
aun no ha       sido recorrido
      if(dist[j] < min && !visited[j]){
        min<-dist[j]
        u<-j
      }
    }
    visited[u] <- 1 #Se registra el nodo u como visitado
    count <- count+1

    for(j in 1:n){
      #Procedimiento de relajación para nodos que aun no han sido
recorridos
      if((dist[j]) > dist[u]+w[u,j] && !visited[j]){
        dist[j]<-dist[u]+w[u,j] #Se actualiza la distancia al destino
        path[j]<-u #Se registra el nodo en la ruta
      }
    }
  }
  dist[length(dist)]
}
```

Se presenta la matriz de pesos entre vértices sobre la cual se implementará el algoritmo

```
w<-matrix(Inf,10,10)
colnames(w)<-c("A","B","C","D","E","F","G","H","I","J")
rownames(w)<-c("A","B","C","D","E","F","G","H","I","J")
w['A','B']<-2
w['A','C']<-4
w['A','D']<-3
w['B','E']<-7
w['B','G']<-6
w['C','E']<-3
w['C','F']<-2
w['C','G']<-4
w['D','E']<-4
w['D','F']<-1
w['D','G']<-5
w['E','H']<-1
w['E','I']<-4
w['F','I']<-3
w['G','H']<-3
w['G','I']<-3
w['H','J']<-3
w['I','J']<-4
w

##     A   B   C   D   E   F   G   H   I   J
## A Inf   2   4   3 Inf Inf Inf Inf Inf Inf
## B Inf Inf Inf Inf   7 Inf   6 Inf Inf Inf
## C Inf Inf Inf Inf   3   2   4 Inf Inf Inf
## D Inf Inf Inf Inf   4   1   5 Inf Inf Inf
## E Inf Inf Inf Inf Inf Inf Inf   1   4 Inf
## F Inf Inf Inf Inf Inf Inf Inf Inf   3 Inf
## G Inf Inf Inf Inf Inf Inf Inf   3   3 Inf
## H Inf Inf Inf Inf Inf Inf Inf Inf Inf   3
## I Inf Inf Inf Inf Inf Inf Inf Inf Inf   4
## J Inf Inf Inf Inf Inf Inf Inf Inf Inf Inf
```

Se prueba el algoritmo para calcular la distancia más corta entre los vértices (A, J)

```
#Prueba con nodo destino J
s<-1
dest<-10
result <- dijkstra(s,w)
print("Distancia más corta entre A y J")

## [1] "Distancia más corta entre A y J"

print(result)

## [1] 11
```