Estructuras de Datos y Algoritmos - IIC2133
Pauta Control 6

1. Dado un string $S = a_1 \cdots a_n$ se define k como el mínimo valor tal que $S = P_1 \cdots P_k$, donde cada P_i es un palíndromo.

a) Escribe la función de recurrencia $K(s, i, j)$ que dado un subtring $s[i{:}j]$ determine el valor del k antes descrito, justificando su correctitud.

Solución:

$$K(s,i,j) = \begin{cases} 1 & \text{si } s[i:j] \text{ es un palíndromo} \\ \min\limits_{x=i}^{j-1}(K(s,i,x) + K(s,x+1,j)) & \text{en otro caso} \end{cases}$$

[0.5pt] por el caso base de la recurrencia

[1pt] por el caso recursivo

[0.5pts] por que los límites del caso recursivo sean correctos. Se perdona hasta 1 error en los límites.

Por ejemplo

Min de x=i … j, o K(s,i,x) + K(s,x,j) son errores en los límites del caso recursivo.

[1pt] Por la justificación de correctitud.

Justificación:

La correctitud del caso base es trivial. **[0pts]**

Lema: existe **alguna** manera de cortar S en dos strings S_1 y S_2 tal que $S = S_1 S_2$ y $k(S) = k(S_1) + k(S_2)$. **[0.25pts]** (Esta propiedad es la que hace que la recurrencia funcione)

No tenemos cómo saber a priori cual es la manera óptima de cortar S, pero por el lema, **alguno** de los cortes entregará el k mínimo. Así que para encontrarlo, probamos **todos** los posibles cortes de S y elegimos el que nos entregue el menor k **[0.25pts]**

[Demostración del lema] **[0.5pts]**

(Para demostrar esto basta con probar que existe al menos un corte que cumple la propiedad del lema)

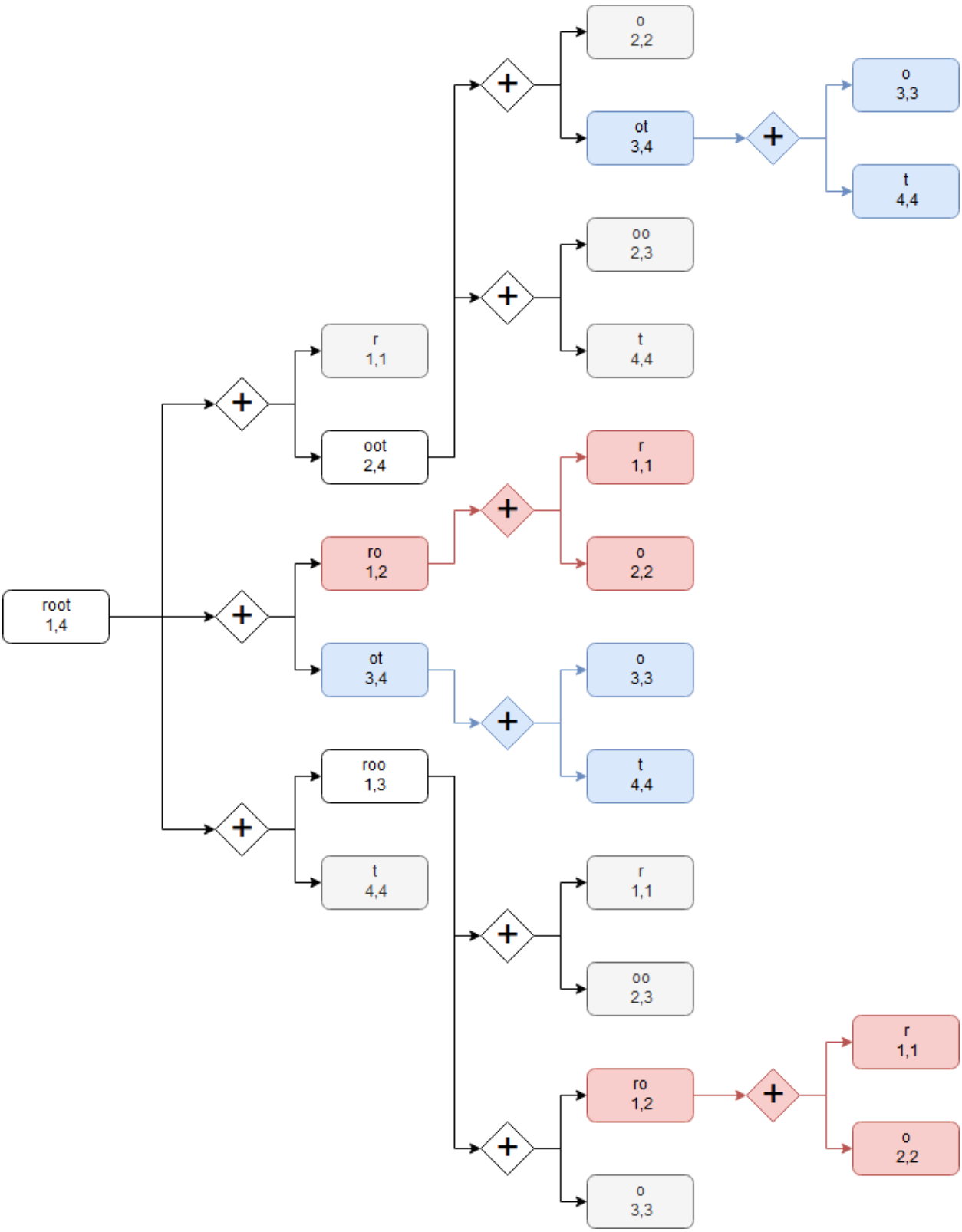
Posible demostración:

Para un string S dado siempre existe una forma de expresarlo como $S = P_1 \cdots P_k$, aunque sea con $k = n$. Si elegimos un y cualquiera entre 1 y $k - 1$, y lo usamos para cortar S en 2 de la siguiente manera: $S_1 = P_1 \cdots P_y$, $S_2 = P_{y+1} \cdots P_k$ tenemos que el k de S_1 es y, y el k de S_2 es $k - y$. Con esto tenemos al menos una forma de cortarlo.

[No es necesario que la justificación sea tan formal, mientras mencionen y justifiquen correctamente los puntos descritos.]

b) Explica y justifica el uso de programación dinámica para este problema. Haz un diagrama de la recursión para respaldar tus argumentos.

Usemos como ejemplo la palabra *root*. (pueden usar la palabra que quieran)


root
1,4
r
1,1
oot
2,4
o
2,2
ot
3,4
o
3,3
t
4,4
oo
2,3
t
4,4
ro
1,2
r
1,1
o
2,2
ot
3,4
o
3,3
t
4,4
roo
1,3
t
4,4
r
1,1
oo
2,3
ro
1,2
r
1,1
o
2,2
o
3,3

[1pt] por mostrar que la recurrencia puede generar un mismo subproblema más de una vez:

Ejemplo: como se ve en el diagrama, las llamadas K(s,1,2) y K(s,3,4) se repiten, y generan un árbol entero repetido. [No es necesario dibujar el diagrama **completo** de la recursión mientras quede claro que se generan subárboles repetidos y los indiquen correctamente]

Sólo si está justificado correctamente, [1pt] por explicar cómo aplicar programación dinámica:

Ejemplo: Podemos calcular el valor de cada llamada una sola vez, guardar su resultado, y las siguientes veces que se haga esa llamada simplemente retornar el valor previamente calculado, así ahorramos generar árboles de llamadas repetidos.

c) **[1pt]** por complejidad con PD:

Con programación dinámica se calculará a lo más una vez cada llamada distinta por lo que la complejidad es el costo de cada llamada una sola vez.

Sabemos que se tiene que verificar si la palabra es un palíndromo por lo que cada llamada toma O(largo sub palabra).

Por lo tanto el tiempo total es la suma de los largos de todas las sub palabras distintas:

$$T(n) = 1 \cdot n + 2 \cdot (n-1) + 3 \cdot (n-2) + \ldots + n \cdot 1$$

$$T(n) \in O(n^3)$$

[0.5pts de bonus] por la complejidad sin programación dinámica:

El peor caso es cuando $k = n$.

Planteamos T(n) como el tiempo de procesar una palabra de largo n

$$T(n) = n + \sum_{x=1}^{n-1} T(x) + T(n-x)$$

Por simetría, eso es equivalente a

$$T(n) = n + 2\sum_{x=1}^{n-1} T(x)$$

Esta función es estrictamente mayor a

$$f(n) = \sum_{x=1}^{n-1} f(x)$$

Desarrollamos:

$$f(n) = f(n-1) + f(n-2) + f(n-3) + \ldots + f(1)$$

$$f(n) = f(n-1) + f(n-1)$$

$$f(n) = 2 \cdot f(n-1)$$

Y esto es $f(n) = 2^{n-1}$

Por lo tanto $T(n) \in \Omega(2^n)$. En otras palabras es al menos exponencial.

Estructuras de Datos y Algoritmos - IIC2133
Pauta Control 7

1) Describe un algoritmo que, dado *n* enteros en el rango 0 a k, preprocesa el input y luego contesta cualquier consulta acerca de cuántos de los *n* enteros caen en un rango [*a … b*] en tiempo O(1). Tu algoritmo debe usar un tiempo de preprocesamiento $\Theta(n + k)$.

   Hint: Recuerda el algoritmo *counting-sort* estudiado en clase que, además de tener un arreglo de entrada y otro de salida, usa internamente un arreglo auxiliar. Más allá de la inicialización de este arreglo auxiliar, *counting-sort* ejecuta tres etapas. La tercera etapa es la que pone los datos del arreglo de entrada ordenadamente en el arreglo de salida; recuerda las dos primeras.

   (4 puntos por preprocesamiento:
   - 1 pto. por crear un array de o a k
   - 1 pto. por contar cuántos de cada número hay
   - 2 ptos. por contar el acumulado del array.)

```
preprocesamiento (data, k, n):
        sea count[0 … k] un nuevo arreglo
        for i = 0 … k:
                count[ i ] = 0
        for j = 1 … n:
                count[data[ j ]] = count[data[ j ]] + 1
        for i = 1 … k:
                count[ i ] = count[ i ] + count[ i - 1]
        return count
```

   (2 puntos por hacer la consulta.)

```
consulta (a, b, count):
        rango = count[ b ] - count[ a - 1 ]
        return rango
```

2)

Mientras expliquen de una forma clara cómo armar el grafo, se consideró correcto. Aquí una propuesta de solución:

Armar un grafo bipartito en que un lado del grafo corresponden a vértices representados como celdas blancas en el tablero y el otro lado corresponda las celdas grises del tablero (también pueden usarse celdas pares e impares según su ubicación dentro del tablero). [1 pto]

Todos los vértices que representan una celda blanca se unen a un único nodo origen del flujo con aristas de costo 1. Todos los vértices que representan una celda gris se unen a un único nodo T, sumidero, con aristas de capacidad 1. No se consideran las celdas negras. [1 pto]

Se puede de esta forma plantear para cada vértice blanco (que representa una celda blanca del tablero) las aristas que representa las celdas grises que se encuentran arriba, abajo, derecha o izquierda del tablero. De esta manera, a partir de un vértice blanco, se genera una arista de capacidad 1 a cada vértice gris. Se puede representar entonces un dominó que se puede colocar entre un vértice blanco y gris, para un problema de flujo máximo. [1 pto]

[3 ptos]

Dado que desde el origen una unidad de flujo puede llegar a cada vértice blanco, esto quiere decir que al menos se puede colocar para cada celda blanca un dominó. A partir de cada vértice se puede pasar esa única unidad de flujo a través de una de las aristas hacia algún nodo gris posible. Dado que para los nodos grises solo se tiene una arista que llega al vértice sumidero, esto obliga a que para una unidad de flujo que pase de un nodo blanco a gris, solo se pueda asignar un único dominó [2 puntos si no se menciona el flujo desde los grises al sumidero]. Se representa a través del flujo máximo la máxima cantidad de dominós que pueden asignarse dado un tablero, dado que representa el problema del tablero y sus restricciones, asignando a lo más 1 dominó por cada celda blanco y gris. [3 ptos]