Estructuras de datos
Clase teórica 5

UNIVERSIDAD NACIONAL DE COLOMBIA
SEDE MEDELLÍN

Contenido

- Ordenamiento en una lista
- El juego de Josephus Flavius

Material elaborado por: Julián Moreno

Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión

Ordenamiento de listas

Pese a que tener una lista ordenada no sirve de mucho, al menos para realizar búsquedas, es posible que para determinados problemas sea necesario o pertinente hacerlo. Si ese fuera el caso, ¿qué podríamos hacer?

Básicamente, igual que con los arreglos, hay dos caminos a seguir:

A. Si ya se tiene la lista, y este está desordenada, se puede ordenar usando un algoritmo de ordenamiento

B. Si se parte de una lista vacía, se puede ir insertando progresivamente cada elemento en el lugar que le corresponde

Algoritmo de ordenamiento bubleSort

```
LinkedList<Integer> L = new LinkedList<Integer>();
Scanner entrada = new Scanner(System.in);
int N = entrada.nextInt();
for (int i = 0; i < N; i++) {
    L.add(entrada.nextInt());
}
int aux;
for (int i = 1; i < N; i++) {
    for (int j = 0; j < N - i; j++) {
        if (L.get(j) > L.get(j + 1)) {
            aux = L.get(j);
            L.set(j, L.get(j+1));
            L.set(j + 1, aux);
        }
    }
}
```

¿Cuál es la eficiencia de este algoritmo?

$f(n)= 3+3N +1+4N(N-1)(6N)/2$, por tanto $O(N^3)$

*Si en vez de usar el indexado, aprovechamos la arquitectura de la lista “moviéndonos” usando iteradores, esta eficiencia podría llegar a reducirse a $O(N^2)$

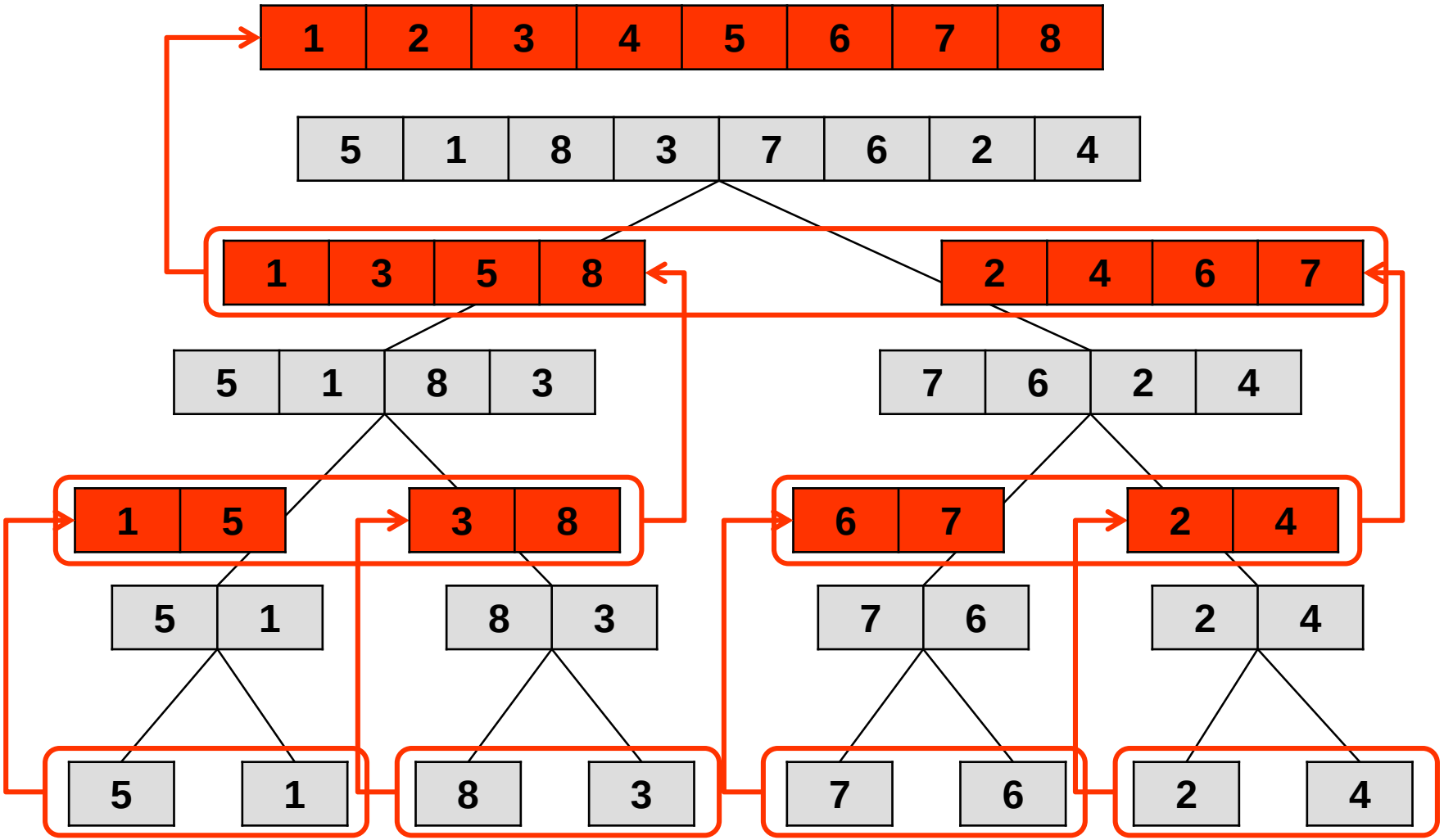
Algoritmo de ordenamiento progresivo

```
LinkedList<Integer> L = new LinkedList<Integer>();
Scanner entrada = new Scanner(System.in);
Iterator<Integer> it;
int j;
int x = 0;
int N = entrada.nextInt();
L.add(entrada.nextInt());
for (int i = 1; i < N; i++) {
    x = entrada.nextInt();
    it = L.listIterator();
    j=0;
    while(it.hasNext()){
        if(it.next() < x)
            j++;
        else
            break;
    }
    L.add(j, x);
}
```

¿Cuál es la eficiencia de este algoritmo?

$f(n) = 7+5*3*N(N-1)/2 + 5N^2$, por tanto $O(N^2)$

Algoritmo de ordenamiento mergeSort



¿Cuál es la eficiencia de este algoritmo en el caso de las listas?

Resulta que, aunque un tanto más complicado de programar que en el caso de arreglos, el mismo *O(N*log(N))*

El juego de Josephus Flavius

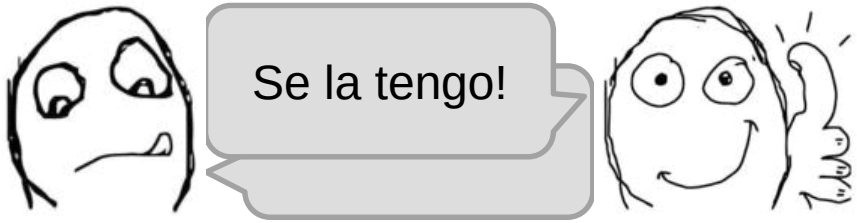
Josephus Flavius fue un famoso historiador judío del siglo I. Durante la guerra judío-romana fue rodeado por el ejército romano y quedó atrapado junto a 40 soldados en una cueva. La leyenda dice que, prefiriendo el suicido a la captura, los soldados decidieron formar un círculo. Luego se numeraron del 1 al 41 (para incluir a Josephus). Comenzando con el soldado 1, contaban 3 soldados a la derecha y ese debía ser asesinado por el compañero de la izquierda. El proceso se repetía a partir de allí hasta que no quedara sino uno en pie (al último no había quien lo matara). Josephus no quería morir y por eso hizo un rápido ejercicio mental, encontró el “puesto seguro” y escapó de la muerte (los romanos al ver la masacre y que era el único sobreviviente lo dejaron con vida).

El juego de Josephus Flavius

Basado en hechos reales …

En su entrevista para una de las más importantes empresas de software del mundo a un estudiante de la nacho le pidieron: “Diseñe un algoritmo para determinar el “puesto seguro” en el juego de Josephus considerando *n* jugadores y un conteo de *k* para eliminar un jugador (*n* y *k* son enteros positivos)”.

Solución de nuestro protagonista:



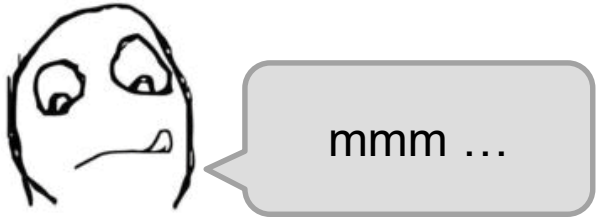
Creamos un arreglo *X* de *n* números enteros, en la posición 0 ponemos el "1", en la 1 el "2" y así sucesivamente hasta "n".

```
posición = 0
cantidadJugadores = n
while(cantidadJugadores > 1){
    contador = 0
    while(contador < k && X[posición] != -1){
        posición++
        if (posición == n-1) posición = 0
        if (X[posición] != -1) contador++
     }
     X[posición] = -1;   cantidadJugadores--
}
```

Al terminar el while más externo buscamos la única posición que no valga -1 y esa es la solución

Objeción del entrevistador: ¿cuál es la eficiencia del algoritmo en términos de *n* y k?, ¿qué pasaría si son muy grandes?

Análisis de nuestro protagonista:

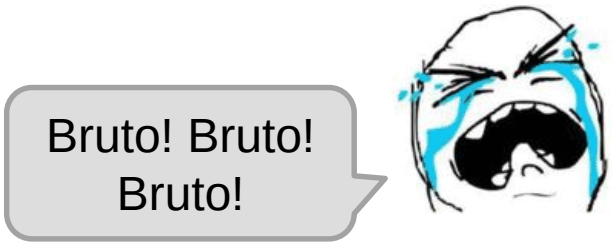


- Como no se puede simplemente saltar en el arreglo *k* posiciones hacia la derecha pues no se sabe cuantos en el medio han muerto, “matar uno” no es *O(1)*

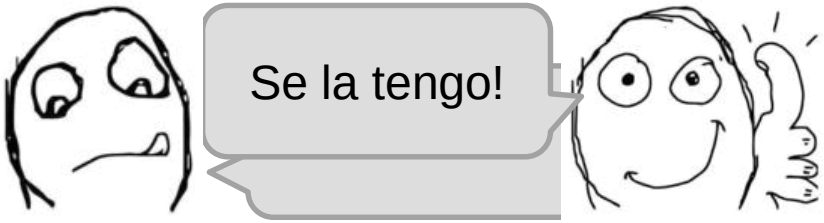
- Encontrar la siguiente posición “no muerto” puede ser *O(n)* en el peor de los casos, por tanto encontrar la posición k-ésima puede ser *O(nk),* y eso es solo para matar uno

- Entonces matar *n-1* tomaría $O(n^2k)$

- Si *n* es por ejemplo un millón y *k* es quinientos mil, eso daría … 500.000.000.000.000.000 (5E+17)



Solución #2 de nuestro protagonista



Creo una lista enlazada *L*, agrego un “1”, luego un “2” y así sucesivamente hasta “n”.

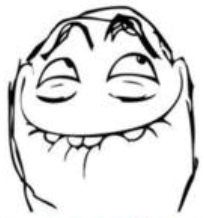
```
posición = 0
cantidadJugadores = n
while(cantidadJugadores > 1){
    contador = 0
    while(contador < k){
        posición++
        if (posición == cantidadJugadores-1) posición = 0
        contador++
     }
     L.remove(posición);   cantidadJugadores--
}
```

Al terminar el while más externo buscamos el único elemento de la lista que quede

Esta vez le voy a hacer el análisis de la eficiencia antes de que me pregunte para que no me corche

Análisis de nuestro protagonista:

- Como ya se trata de una lista donde solo están jugadores vivos matar al *k* siguiente implica: el while más interno que es *O(k)* mas el remove que es *O(n)*
- Entonces matar *n-1* tomaría *O(n(n+k)) …* Si, otra vez, *n* es por ejemplo un millón y *k* es quinientos mil, eso daría … 1.500.000.000.000 (1,5E+12)

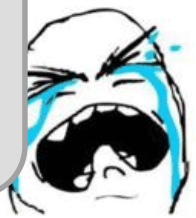


Bien!, ya le bajé un montón de ceros

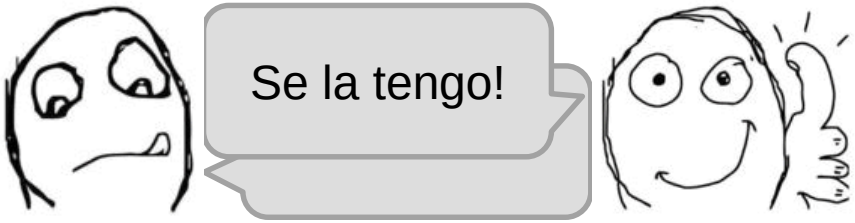
Objeción del entrevistador: ¿No se puede hacer mejor considerando que si en algún momento *k* es mayor que la cantidad de jugadores con vida estaría dando vueltas "innecesarias"?

mmm … tiene razón, si por ejemplo no quedan sino 10 jugadores y *k* es 25 habría que dar dos vueltas y media, en vez de simplemente moverse 5 posiciones desde donde esté parado. Si *k* es mucho más grande que *n*, peor aún!

Bruto!



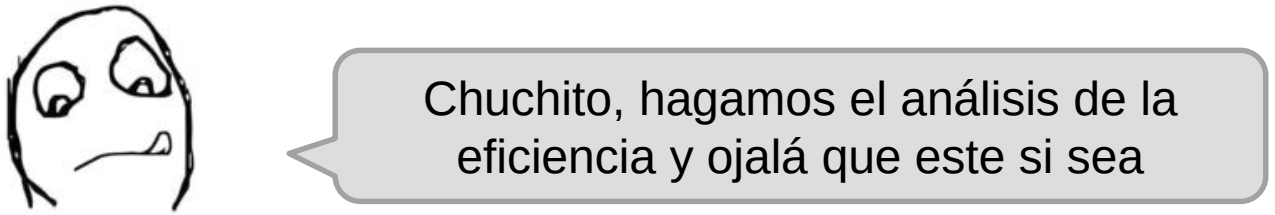
Solución #3 de nuestro protagonist



Creo una lista enlazada *L*, agrego un “1”, luego un “2” y así sucesivamente hasta “n”.

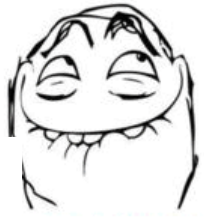
```
posición = 0
cantidadJugadores = n
while(cantidadJugadores > 1){
    contador = 0
    while(contador < k%cantidadJugadores){
        posición++
        if (posición == cantidadJugadores-1) posición = 0
        contador++
    }
    L.remove(posición);   cantidadJugadores--
}
```

Al terminar el while más externo buscamos el único elemento de la lista que quede



Análisis de nuestro protagonista:

- Usando una lista y moviéndose por las posiciones usando el modulo garantizo que, si en algún momento *k* es mayor a la cantidad de jugadores, habría que moverse, a lo sumo, la cantidad de jugadores menos 1
- Siendo así, $O(n(n+k))$ se convierte en $O(n^2)$



En otras palabras: ya no importa que *k* sea grande. Si por ejemplo *n* es 20 y *k* es un millón, eso daría solo 400, comparado con los 20 millones de la solución anterior

Entrevistador: muchas gracias por venir … lo estaremos llamando

Pero, lo hice bien, ¿no cierto?