Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2019

Departamento de Computación - FCEyN - UBA

Algoritmos de ordenamiento sobre secuencias

Ordenamiento de secuencias

- proc *ordenar*(inout $s : seq\langle\mathbb{Z}\rangle$){
  Pre $\{s = S_0\}$
  Post $\{mismos(s, S_0) \ \wedge \ ordenado(s)\}$
  }
- pred *mismos*$(s, t : seq\langle\mathbb{Z}\rangle)${
  $(\forall e : \mathbb{Z})(\#\text{apariciones}(s, e) = \#\text{apariciones}(t, e))$
  }
- aux $\#\text{apariciones}(s : seq\langle T\rangle, e : T) : \mathbb{Z} =$
  $\sum_{i=0}^{|s|-1}(\text{if } s[i] = e \text{ then 1 else 0 fi})$
- pred *ordenado*$(s : seq\langle\mathbb{Z}\rangle)${
  $(\forall i : \mathbb{Z})(0 \leq i < |s| - 1 \rightarrow_L s[i] \leq s[i + 1])$
  }

Ordenamiento de secuencias

- Modificamos la secuencia solamente a través de intercambios de elementos.

proc *swap*(inout $s : seq\langle\mathbb{Z}\rangle$, in $i, j : \mathbb{Z}$){
  Pre $\{0 \leq i, j < |s| \wedge s = S_0\}$
  Post $\{s[i] = S_0[j] \wedge s[j] = S_0[i] \wedge$
  $(\forall k : \mathbb{Z})(0 \leq k < |s| \wedge i \neq k \wedge j \neq k \rightarrow_L s[k] = S_0[k])\}$
}

- **Propiedad 1:**

$$s = S_0 \rightarrow mismos(s, S_0)$$

- **Propiedad 2:**

$$\{mismos(s, S_0)\}$$
$$\text{swap(s,i,j)}$$
$$\{mismos(s, S_0)\}$$

- De esta forma, nos aseguramos que $mismos(s, S_0)$ a lo largo de la ejecución del algoritmo.

Ordenamiento por selección (Selection Sort)

- **Idea:** Seleccionar el mínimo elemento e intercambiarlo con la primera posición de la secuencia. Repetir con el segundo, etc.

```
void selectionSort(vector<int> &s) {
  for(int i=0; i<s.size(); i++) {
    // indice del minimo elemento de s entre i y s.size()
    int minPos = ◦ ..
    swap(s, i, minPos);
  }
}
```

Ordenamiento por selección (Selection Sort)

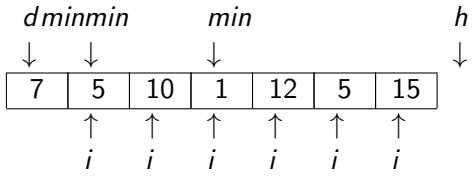
- Podemos refinar un poco el código:

```
void selectionSort(vector<int> &s) {
  for(int i=0; i<s.size()-1; i++) {
    int minPos= findMinPosition(s, i, s.size());
    swap(s, i, minPos);
  }
}
```

- Entonces surge la necesidad de especificar el problema auxiliar de buscar el mínimo entre i y $s.size()$:

proc *findMinPosition*(in $s : seq\langle\mathbb{Z}\rangle$, in $d, h : \mathbb{Z}$, out $min : \mathbb{Z}$){
  Pre $\{0 \leq d < h \leq |s|\}$
  Post $\{d \leq min < h$
    $\wedge_L\ (\forall i : \mathbb{Z})(d \leq i < h \rightarrow_L s[min] \leq s[i])\}$
}

Buscar el Mínimo Elemento



- ¿Qué invariante de ciclo podemos proponer?

$$d \leq min < i \leq h \wedge_L (\forall j : \mathbb{Z})(d \leq j < i \rightarrow_L s[min] \leq s[j])$$

- ¿Qué función variante podemos usar?

$$fv = h - i$$

Buscar el Mínimo Elemento

- Invariante:

$$d \leq min < i \leq h \ \wedge_L (\forall j : \mathbb{Z})(d \leq j < i \rightarrow_L s[min] \leq s[j])$$

- Función variante

$$fv = h - i$$

- ¿Cómo lo implementamos?

```
int findMinPosition(vector<int> &s, int d, int h) {
  int min = d;
  for(int i = d + 1; i < h; i++) {
    if (s[i] < s[min]) {
      min = i;
    }
  }
  return min;
}
```


Recap: Teorema de corrección de un ciclo

- **Teorema.** Sean un predicado I y una función $fv : \mathbb{V} \rightarrow \mathbb{Z}$ (donde $\mathbb{V}$ es el producto cartesiano de los dominios de las variables del programa), y supongamos que $I \Rightarrow \text{def}(B)$. Si

1. $\mathbb{P}_\mathbb{C} \Rightarrow \mathbb{I}$,
2. $\{\mathbb{I} \wedge \mathbb{B}\}\ \mathbb{S}\ \{\mathbb{I}\}$,
3. $\mathbb{I} \wedge \neg\mathbb{B} \Rightarrow \mathbb{Q}_\mathbb{C}$,
4. $\{\mathbb{I} \wedge \mathbb{B} \wedge V_0 = fv\}$ **S** $\{fv < V_0\}$,
5. $\mathbb{I} \wedge fv \leq 0 \Rightarrow \neg\mathbb{B}$,

... entonces la siguiente tripla de Hoare es válida:

$$\{\mathbb{P}_\mathbb{C}\}\ \textbf{while B do S endwhile}\ \{\mathbb{Q}_\mathbb{C}\}$$

Buscar el Mínimo Elemento

- $\mathbb{P}_\mathbb{C} \equiv 0 \leq d < h \leq |s| \wedge min = d \wedge i = d + 1$
- $\mathbb{Q}_\mathbb{C} \equiv d \leq min < h$
  $\wedge_L (\forall i : \mathbb{Z})(d \leq i < h \rightarrow_L s[min] \leq s[i])$
- $\mathbb{B} \equiv i < h$
- $\mathbb{I} \equiv d \leq min < i \leq h$
  $\wedge_L (\forall j : \mathbb{Z})(d \leq j < i \rightarrow_L s[min] \leq s[j])$
- $fv = h - i$

```
int findMinPosition(vector<int> &s, int d, int h) {
  int min = d;
  for(int i=d+1; i<h; i++) {
    if (s[i] < s[min]) {
      min = i;
    }
  }
  return min;
}
```


Correctitud: Buscar el Mínimo Elemento

- $\mathbb{P}_\mathbb{C} \equiv 0 \leq d < h \leq |s| \wedge min = d \wedge i = d + 1$
- $\mathbb{Q}_\mathbb{C} \equiv d \leq min < h$
  $\wedge_L (\forall i : \mathbb{Z})(d \leq i < h \rightarrow_L s[min] \leq s[i])$
- $\mathbb{B} \equiv i < h$
- $\mathbb{I} \equiv d \leq min < i \leq h$
  $\wedge_L (\forall j : \mathbb{Z})(d \leq j < i \rightarrow_L s[min] \leq s[j])$
- $fv = h - i$

- ¿$\mathbb{I}$ es se cumple al principio del ciclo (punto 1.)? ✓
- ¿Se cumple la postcondición del ciclo a la salida del ciclo (punto 3.)? ✓
- ¿Si la función variante alcanza la cota inferior la guarda se deja de cumplir (punto 5.)? ✓

Correctitud: Buscar el Mínimo Elemento

- $\mathbb{I} \equiv d \leq min < i \leq h$
  $\wedge_L (\forall j : \mathbb{Z})(d \leq j < i \rightarrow_L s[min] \leq s[j])$
- $fv = h - i$

```
int findMinPosition(vector<int> &s, int d, int h) {
  int min = d;
  for(int i=d+1; i<h; i++) {
    if (s[i] < s[min]) {
      min = i;
    }
  }
  return min;
}
```

- ¿$\mathbb{I}$ se preserva en cada iteración (punto 2.)? ✓
- ¿La función variante es estrictamente decreciente (punto 4.)?✓

Ordenamiento por selección (Selection Sort)

- Volvamos ahora al programa de ordenamiento por selección:

```
void selectionSort(vector<int> &s) {
  for(int i=0; i<s.size(); i++) {
    int minPos = findMinPosition(s, i, s.size());
    swap(s, i, minPos);
  }
}
```

- $\mathbb{P}_\mathbb{C} \equiv i = 0 \wedge s = S_0$
- $\mathbb{Q}_\mathbb{C} \equiv mismos(s, S_0) \wedge \ ordenado(s)$
- $\mathbb{B} \equiv i < |s|$
- $\mathbb{I} \equiv$?
  - ¡Luego de la i-ésima iteración, $subseq(s, 0, i)$ contiene los i primeros elementos ordenados! ¿Tenemos entonces el invariante del ciclo?
  - $\mathbb{I} \equiv mismos(s, S_0) \wedge ((0 \leq i \leq |s|) \wedge_L ordenado(subseq(s, 0, i)))$
- $fv = |s| - i$

Ordenamiento por selección (Selection Sort)

- $\mathbb{I} \equiv \mathit{mismos}(s, S_0) \wedge ((0 \leq i \leq |s|) \wedge_L \mathit{ordenado}(\mathit{subseq}(s, 0, i)))$
- $\mathit{fv} = |s| - i$

```
void selectionSort(vector<int> &s) {
  for(int i=0; i<s.size(); i++) {
    int minPos = findMinPosition(s, i, s.size());
    swap(s, i, minPos);
  }
}
```

- ¿$\mathbb{I}$ se preserva en cada iteración (punto 2.)? ✗
- Contraejemplo:
  - Si arrancamos la iteración con $i = 1$ y $s = \langle 100, 2, 1 \rangle$
  - Terminamos con $i = 2$ y $s = \langle 100, 1, 2 \rangle$ que no satisface I

Debemos reforzar el invariante para probar la corrección:

$$I \equiv \mathit{mismos}(s, S_0) \wedge ((0 \leq i \leq |s|) \wedge_L (\mathit{ordenado}(\mathit{subseq}(s, 0, i))) \wedge (\forall j, k : \mathbb{Z})((0 \leq j < i \wedge i \leq k < |s|) \rightarrow_L s[j] \leq s[k]))$$

Correctitud: Ordenamiento por selección (Selection Sort)

$$\mathbb{I} \equiv \mathit{mismos}(s, S_0) \wedge ((0 \leq i \leq |s|) \wedge_L (\mathit{ordenado}(\mathit{subseq}(s, 0, i))) \wedge (\forall j, k : \mathbb{Z})((0 \leq j < i \wedge i \leq k < |s|) \rightarrow_L s[j] \leq s[k]))$$

Gráficamente:

$x \in \mathit{subseq}(s, 0, i)$	$y \in \mathit{subseq}(s, i, \lvert s \rvert)$
$\leq y$	$\geq x$
ordenado	?

Correctitud: Ordenamiento por selección (Selection Sort)

- $\mathbb{P}_\mathbb{C} \equiv i = 0 \wedge s = S_0$
- $\mathbb{Q}_\mathbb{C} \equiv \mathit{mismos}(s, S_0) \wedge \ \mathit{ordenado}(s)$
- $\mathbb{B} \equiv i < |s|$
- $\mathbb{I} \equiv \mathit{mismos}(s, S_0) \wedge ((0 \leq i \leq |s|) \wedge_L (\mathit{ordenado}(\mathit{subseq}(s, 0, i))) \wedge (\forall j, k : \mathbb{Z})((0 \leq j < i \wedge i \leq k < |s|) \rightarrow_L s[j] \leq s[k]))$
- $\mathit{fv} = |s| - i$

- ¿$\mathbb{I}$ es se cumple al principio del ciclo (punto 1.)? ✓
- ¿Se cumple la postcondición del ciclo a la salida del ciclo (punto 3.)?✓
- ¿Si la función variante alcanza la cota inferior la guarda se deja de cumplir (punto 5.)?✓

Correctitud: Ordenamiento por selección (Selection Sort)

- $\mathbb{I} \equiv \mathit{mismos}(s, S_0) \wedge ((0 \leq i \leq |s|) \wedge_L (\mathit{ordenado}(\mathit{subseq}(s, 0, i))) \wedge (\forall j, k : \mathbb{Z})((0 \leq j < i \wedge i \leq k < |s|) \rightarrow_L s[j] \leq s[k]))$
- $\mathit{fv} = |s| - i$

```
void selectionSort(vector<int> &s) {
  for(int i=0; i<s.size(); i++) {
    int minPos = findMinPosition(s, i, s.size());
    swap(s, i, minPos);
  }
}
```

- ¿$\mathbb{I}$ se preserva en cada iteración (punto 2.)? ✓
- ¿La función variante es estrictamente decreciente (punto 4.)?✓

Ordenamiento por selección (Selection Sort)

```
int findMinPosition(vector<int> &s, int d, int h) {
  int min = d;
  for(int i=d+1; i<h; i++) {
    if (s[i] < s[min]) {
      min = i;
    }
  }
  return min;
}
void selectionSort(vector<int> &s) {
  for(int i=0; i<s.size(); i++) {
    int minPos = findMinPosition(s,i,s.size());
    swap(s, i, minPos);
  }
}
```

- ¿Cómo se comporta este algoritmo?
- Veámoslo en `https://visualgo.net/es/sorting`.

Tiempo de ejecución de peor caso

findMinPosition

- Sea $n = |s|$ ¿cuál es el tiempo de ejecución de peor caso de `findMinPosition`?

`int min = d;`	c_1	1
`for(int i=d+1; i<h; i++) {`	c_2	n
`if (s[i] < s[min]) {`	c_3	$n-1$
`min = i;`	c_4	$n-1$
`}`		
`}`		
`return min;`	c_5	1

- $T_{findMinPosition}(n) = 1*c_1+n*c_2+(n-1)*c_3+(n-1)*c_4+1*c_5$
- $T_{findMinPosition}(n) \in O(n)$
- Decimos que `findMinPosition` tiene un tiempo de ejecución de peor caso lineal en función de la longitud de la secuencia.

Tiempo de ejecución de peor caso

selectionSort

- Sea $n = |s|$ ¿cuál es el tiempo de ejecución de peor caso para el programa `selectionSort`?

`for(int i=0; i<s.size(); i++) {`	c'_1	$n+1$
`int minPos=findMinPosition(s,i,s.size());`	$c'_2 * n$	n
`swap(s, i, minPos);`	c'_3	n
`}`		

- $T_{selectionSort}(n) = (n+1) * c'_1 + n * n * c'_2 + n * c'_3$
- $T_{selectionSort}(n) \in O((n+1) * n) = O(n^2 + n) = O(n^2)$
- Decimos que `selectionSort` tiene un tiempo de ejecución de peor caso cuadrático en función de la longitud de la secuencia.

Ordenamiento por selección (Selection Sort)

- Variantes del algoritmo básico:
  1. Cocktail sort: consiste en buscar en cada iteración el máximo y el mínimo del vector por ordenar, intercambiando el mínimo con i y el máximo con $|s| - i - 1$.
  2. Bingo sort: consiste en ubicar todas las apariciones del valor mínimo en el vector por ordenar, y mover todos los valores mínimos al mismo tiempo (efectivo si hay muchos valores repetidos).
- El tiempo de ejecución de peor caso de ambas variantes en función de $n = |s|$ es:
  - $T_{cocktailSort}(n) \in O(n^2)$
  - $T_{bingoSort}(n) \in O(n^2)$
- Por lo tanto, ambas variantes de `selectionSort` tienen el *"mismo"* tiempo de ejecución de peor caso (cuadrático)

Intervalo

Break!

Ordenamiento por inserción (Insertion Sort)

- Veamos otro algoritmo de ordenamiento, pero donde el invariante (a diferencia de `selectionSort`) es:

$$\mathbb{I} \equiv \mathit{mismos}(s, S_0) \wedge (0 \leq i \leq |s| \ \wedge_L \ \mathit{ordenado}(\mathit{subseq}(s, 0, i)))$$

- Esto implica que en cada iteración los primeros i elementos están ordenados, sin ser necesariamente los i elementos más pequeños del vector.
- La función variante de este algoritmo de ordenamiento (al igual que `selectionSort`) es:

$$fv = |s| - i$$

Ordenamiento por inserción (Insertion Sort)

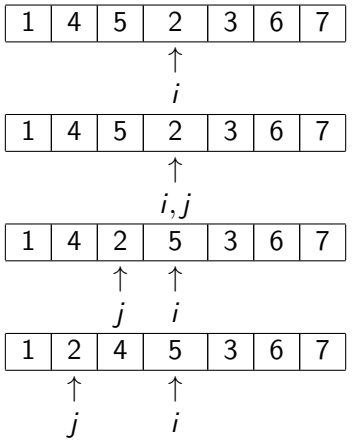
$$\mathbb{I} \equiv \mathit{mismos}(s, S_0) \wedge (0 \leq i \leq |s| \ \wedge_L \ \mathit{ordenado}(\mathit{subseq}(s, 0, i)))$$

```
void insertionSort(vector<int> &s) {
  for(int i=0; i<s.size(); i++) {
    // Tenemos que preservar el invariante ∘ ..
  }
}
```

- ¿$\mathbb{I}$ es se cumple al principio del ciclo (punto 1.)? ✓
- ¿Se cumple la postcondición del ciclo a la salida del ciclo (punto 3.)? ✓
- ¿$\mathbb{I}$ se preserva en cada iteración (punto 2.)?
  - Sabiendo que los primeros i elementos están ordenados, tenemos que hacer que los primeros $i + 1$ elementos pasen a estar ordenados!
  - ¿Cómo lo podemos hacer?

Ordenamiento por inserción (Insertion Sort)

Necesitamos desplazar $s[i]$ hasta una posición donde subseq(s,0,i) esté ordenada de vuelta. Ejemplo, ya están ordenadas las primeras 3 posiciones.



Ordenamiento por inserción (Insertion Sort)

- Antes de comenzar el desplazamiento, tenemos que:

ordenada	
	?
$subseq(s, 0, i)$	$subseq(s, i, \|s\|)$

- Durante el desplazamiento, se cumple que que:

ordenada		ordenada	
	x	$> x$	?
$subseq(s, 0, j)$	$s[j]$	$subseq(s, j+1, i+1)$	$subseq(s, i+1, \|s\|)$

- Al finalizar el desplazamiento, nuevamente tenemos que:

ordenada	
	?
$subseq(s, 0, i+1)$	$subseq(s, i+1, \|s\|)$

Ordenamiento por inserción (Insertion Sort)

- Llamemos `insert` a la función auxiliar que desplaza el elemento s[i] ¿cuál es el invariante para esta función?

$$
\begin{aligned}
\mathbb{I} \equiv\ & 0 \le j \le i \\
\wedge\ & mismos(subseq(s, 0, i+1), subseq(S_0, 0, i+1)) \\
\wedge\ & subseq(s, i+1, |s|) = subseq(S_0, i+1, |s|) \\
\wedge\ & ordenado(subseq(s, 0, j)) \ \wedge\ ordenado(subseq(s, j, i+1)) \\
\wedge\ & (\forall k : \mathbb{Z})(j < k \le i \rightarrow_L s[j] < s[k])
\end{aligned}
$$

- ¿Cuál es la función variante de `insert`?

$$fv = j$$

Ordenamiento por inserción (Insertion Sort)

- ¿Cuál es una posible implementación de `insert`?

```
void insert(vector<int> &s, int i) {
  for(int j=i; j>0 && s[j] < s[j-1]; j--) {
    swap(s, j, j-1);
  }
}
```

- ¿Cuál es una posible implementación de `insertSort`?

```
void insertionSort(vector<int> &s) {
  for(int i=0; i<s.size(); i++) {
    insert(s,i);
  }
}
```

- ¿Cómo se comporta este algoritmo de ordenamiento?
- Veámoslo en `https://visualgo.net/es/sorting`.

Tiempo de ejecución de peor caso

insert

- Sea $n = |s|$ ¿cuál es el tiempo de ejecución de peor caso de `insert`?

`for(int j=i; j>0 && s[j] < s[j-1]; j--) {`	c''_1	$n+1$
`  swap(s, j, j-1);`	c''_2	n
`}`		

- $T_{insert}(n) = c''_1 * (n+1) + c''_2 * n$
- $T_{insert}(n) \in O(n)$
- `insert` tiene tiempo de ejecución de peor caso lineal.

Tiempo de ejecución de peor caso

insertSort

- Sea $n = |s|$ ¿cuál es el tiempo de ejecución de peor caso de `insertSort`?

`for(int i=0; i<s.size(); i++) {`	c_1'''	$n+1$
`  insert(s, i);`	$c_2''' * n$	n
`}`		

- $T_{insertSort}(n) = c_1''' * (n+1) + c_2''' * n * n$
- $T_{insertSort}(n) \in O(n^2)$
- `insertSort` tiene tiempo de ejecución de peor caso cuadrático (igual que `selectionSort`)

Dutch National Flag Problem

Dado una secuencia que contiene colores (rojo, blanco y azul) ordenarlos de modo que respeten el orden de la bandera holandesa (primero rojo, luego blanco y luego azul)

Por ejemplo, si la secuencia es:

$$\langle White, Red, Blue, Blue, Red \rangle$$

El programa debe modificar la secuencia para que quede:

$$\langle Red, Red, White, Blue, Blue \rangle$$

Dutch National Flag Problem

- Si Red=0,White=1 y Blue=2, ¿Cuál sería la especificación del problema?
- proc $dutchNationalFlag$(inout $s : seq\langle\mathbb{Z}\rangle$){
  Pre $\{s = S_0 \wedge (\forall e : \mathbb{Z})(e \in s \leftrightarrow (e = 0 \vee e = 1 \vee e = 2))\}$
  Post $\{mismos(s, S_0) \ \wedge \ ordenado(s)\}$
  }
- ¿Cómo podemos implementar una solución a este problema?
  - ¿Podemos usar algún algoritmo de ordenamiento que conozcamos? **Rta:** podemos usar `insertionSort` o `selectionSort`.
  - ¿Cuál es tiempo de ejecución de peor caso? **Rta:** $T_{dutchNationalFlag}(n) \in O(|s|^2)$
  - ¿Podemos buscar otra solución que tenga un tiempo de ejecución de peor caso lineal?

Dutch National Flag Problem

1. Recorro la secuencia contando la cantidad de apariciones de cada color: RED (0), WHITE (1) y BLUE (2), almacenándolas en una secuencia de tres elementos.
2. Modifico la secuencia usando las cantidades leídas almacenadas en la secuencia de tres elementos.

Dutch National Flag Problem

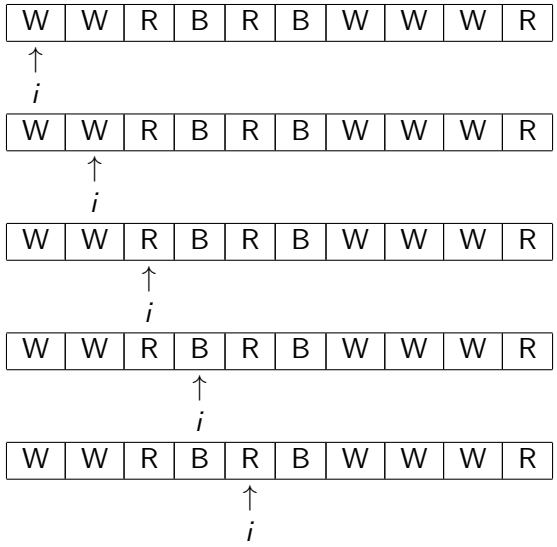
```
#define RED     0
#define WHITE   1
#define BLUE    2

void dutchNationalFlag(vector<int> &s) {
  // contamos la cantidad de apariciones de cada color
  vector<int> colorCount = fillColorCount(s);

  // usamos la cantidad de apariciones para repoblar
  // la secuencia
  populate(s,colorCount);
}
```

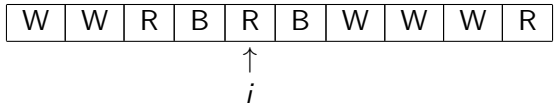

Dutch National Flag Problem

¿Qué invariante de ciclo podemos usar para contar la cantidad de apariciones de cada color?



Dutch National Flag Problem

¿Qué invariante de ciclo podemos usar para contar la cantidad de apariciones de cada color?



$$\mathbb{I} \equiv (0 \leq i \leq |s|) \wedge_L$$

$$(colorCount[RED] = \#apariciones(subseq(s, 0, i), RED) \wedge$$

$$colorCount[WHITE] = \#apariciones(subseq(s, 0, i), WHITE) \wedge$$

$$colorCount[BLUE] = \#apariciones(subseq(s, 0, i), BLUE))$$

¿Qué función variante podemos usar en la primer pasada?

$$fv \equiv |s| - i$$

Dutch National Flag Problem

$$\mathbb{I} \equiv (0 \leq i \leq |s|) \wedge_L$$

$$(colorCount[RED] = \#apariciones(subseq(s, 0, i), RED) \wedge$$

$$colorCount[WHITE] = \#apariciones(subseq(s, 0, i), WHITE) \wedge$$

$$colorCount[BLUE] = \#apariciones(subseq(s, 0, i), BLUE))$$

$$fv \equiv |s| - i$$

¿Cómo podemos implementar `fillColorCount` respetando este invariante y función variante?

```
vector<int> fillColorCount(vector<int> &s) {
  // COMPLETAR
}
```

Dutch National Flag Problem

```
vector<int> fillColorCount(vector<int> &s) {
  vector<int> colorCount = {0,0,0}:
  for(int i=0; i<s.size(); i++) {
    if (s[i]==RED) {
      colorCount[RED]++;
    } else if (s[i]==WHITE) {
      colorCount[WHITE]++;
    } else {
      colorCount[BLUE]++;
    }
  }
  return colorCount;
}
```

Sea $n = |s|$, ¿cuál es el tiempo de ejecución de peor caso del programa `fillColorCount`?

Dutch National Flag Problem

- Sea $n = |s|$, ¿cuál es el tiempo de ejecución de peor caso del programa `fillColorCount`?

```
vector<int> fillColorCount(vector<int> &s) {
  vector<int> colorCount = {0,0,0}; // O(1)
  for(int i=0; i<s.size(); i++) { // O(1)
    if (s[i]==RED) { // O(1)
      colorCount[RED]++; // O(1)
    } else if (s[i]==WHITE) { // O(1)
      colorCount[WHITE]++; // O(1)
    } else { // O(1)
      colorCount[BLUE]++; // O(1)
    }
  }
  return colorCount; // O(1)
}
```

- El `for` se ejecuta $n + 1$ veces
- Por lo tanto, $T_{fillColorCount}(n) \in O(n + 1) = O(n)$

Dutch National Flag Problem

Ahora nos falta repoblar la secuencia con los valores. ¿Qué invariante de ciclo tendríamos que respetar?

$$\begin{aligned}\mathbb{I} \equiv ((0 \leq i \leq |s|) \wedge_L \\ (ordenado(subseq(s, 0, i)) \wedge \\ \#a...(S_0, RED) = \#a...(subseq(s, 0, i), RED) + colorCount[RED]) \wedge \\ \#a...(S_0, WHITE) = \#a...(subseq(s, 0, i), WHITE) + colorCount[WHITE]) \wedge \\ \#a...(S_0, BLUE) = \#a...(subseq(s, 0, i), BLUE) + colorCount[BLUE])))\end{aligned}$$

(#a... quiere decir #apariciones)

¿Qué función variante podemos usar en la primer pasada?

$$fv \equiv |s| - i$$

Dutch National Flag Problem

Sea el siguiente invariante:

$$\begin{aligned}\mathbb{I} \equiv ((0 \leq i \leq |s|) \wedge_L \\ (ordenado(subseq(s, 0, i)) \wedge \\ \#a...(S_0, RED) = \#a...(subseq(s, 0, i), RED) + colorCount[RED]) \wedge \\ \#a...(S_0, WHITE) = \#a...(subseq(s, 0, i), WHITE) + colorCount[WHITE]) \wedge \\ \#a...(S_0, BLUE) = \#a...(subseq(s, 0, i), BLUE) + colorCount[BLUE])))\end{aligned}$$

Y La función variante:

$$fv \equiv |s| - i$$

Escribamos el programa `populate`:

```
void populate(vector<FlagColor> &s, vector<int> &colorCount) {
  //COMPLETAR
}
```

Dutch National Flag Problem

```
void populate(vector<FlagColor> &s, vector<int> &colorCount) {

  for(int i=0; i<s.size(); i++) {
    if (colorCount[RED]>0) {
      s[i] = RED;
      colorCount[RED]—;
    } else if (colorCount[WHITE]>0) {
      s[i] = WHITE;
      colorCount[WHITE]—;
    } else {
      s[i] = BLUE;
      colorCount[BLUE]—;
    }
 }
}
```

Sea $n = |s|$, ¿cuál es el tiempo de ejecución de peor caso de `populate`?

Dutch National Flag Problem

- Sea $n = |s|$, ¿cuál es el tiempo de ejecución de peor caso de `populate`?

```
void populate(vector<FlagColor> &s, vector<int> &colorCount) {
  for(int i=0; i<s.size(); i++) { // O(1)
    if (colorCount[RED]>0) { // O(1)
      s[i] = RED; // O(1)
      colorCount[RED]—; // O(1)
    } else if (colorCount[WHITE]>0) { // O(1)
      s[i] = WHITE; // O(1)
      colorCount[WHITE]—; // O(1)
    } else {
      s[i] = BLUE; // O(1)
      colorCount[BLUE]—; // O(1)
    }}}
```

- El único `for` se ejecuta $n + 1$ veces y las otras operaciones cuestan $O(1)$
- Por lo tanto $T_{populate}(n) \in O(n + 1) = O(n)$

Dutch National Flag Problem

- Ahora volvemos al programa prinicipal conociendo `fillColorCount` y `populate`.

```
void dutchNationalFlag(vector<int> &s) {
  vector<int> colorCount = fillColorCount(s); // O(n)
  populate(s,colorCount); // O(n)
}
```

- ¿Cuál es el tiempo de ejecución de peor caso del programa `dutchNationalFlag` ?
- $T_{dutchNationalFlag} \in O(n + n) = O(n)$
- En el peor caso, la implementación es más eficiente que usar selectionSort o insertionSort

Eficiencia de los Algoritmos de ordenamiento

- Tanto selection sort como insertion sort son algoritmos **cuadráticos** (iteran una cantidad cuadrática de veces)
- ¿Hay algoritmos con comportamiento más eficiente en peor caso?
  - Quicksort y BubbleSort: Peor caso: $O(n^2)$
  - Mergesort y Heapsort: Peor caso: $O(n * log(n))$
  - Counting sort (para secuencias de enteros acotados). Peor caso: $O(n)$
  - Radix sort (para secuencias de enteros). Peor caso: $O(2^{32}) = O(1)$

$$O(1) < O(n) < O(n * log(n)) < O(n^2)$$

- Bubble sort está en la práctica 9 (búsqueda y ordenamiento). El resto los van a ver en **algo2**.

Yapa - La complejidad de la comparación

```
int findMinPosition(...) {
  int min = d; // c1
  for(int i=d+1; i<h;i++){// c2*n
    if (s[i] < s[min]) { // c3*n
      min = i;           // c4*n
    }
  }
  return min;  // c5
} // O(n)
```

```
void selectionSort(...) {
  for(int i=0;i<s.size()-1;i++){// k1*n
    int minP=findMinP(...); //
(k2*n)*n
    swap(s, i, minP); // k3
  }
} // O(n^2)
```

Si en lugar de tomar tiempo constante (O(1)), la operación de comparación "<" fuera lineal (O(n)), entonces... el selectionSort "tardaría" O(n^3). ¿En qué caso podría darse esto?

Bibliografía

- Vickers et al. - Reasoned Programming
  - 6.5 - Insertion Sort
- NIST- Dictionary of Algorithms and Data Structures
  - Selection Sort - https://xlinux.nist.gov/dads/HTML/selectionSort.html
  - Bingo Sort - https://xlinux.nist.gov/dads/HTML/bingosort.html
  - Cocktail Sort - https://xlinux.nist.gov/dads/HTML/bidirectionalBubbleSort.html
- Cormen et al. - Introduction to Algorithms
  - Chapter 2.1 - Insertion Sort