

The transition system Commands

Transition system Commands

Configurations: $\{[C, \sigma] \mid C \text{ a command-sequence, } \sigma \text{ a state}\}$

$[x \leftarrow e; C', \sigma]$	\triangleright	$[C', \sigma \langle x:e \rangle]$	
$[\text{if } b \text{ then } C_1 \text{ else } C_2; C', \sigma]$	\triangleright	$[C_1; C', \sigma]$	if $\sigma(b) = \text{true}$
$[\text{if } b \text{ then } C_1 \text{ else } C_2; C', \sigma]$	\triangleright	$[C_2; C', \sigma]$	if $\sigma(b) = \text{false}$
$[\text{while } b \text{ do } C; C', \sigma]$	\triangleright	$[C; \text{while } b \text{ do } C; C', \sigma]$	if $\sigma(b) = \text{true}$
$[\text{while } b \text{ do } C; C', \sigma]$	\triangleright	$[C', \sigma]$	if $\sigma(b) = \text{false}$

Euclid's algorithm

```
Algorithm Euclid( $m, n$ )  
Input      :  $m, n \geq 1$   
Output     :  $r = \text{gcd}(m_0, n_0)$   
Method     : while  $m \neq n$  do  
              if  $m > n$  then  
                   $m \leftarrow m - n$   
              else  
                   $n \leftarrow n - m;$   
               $r \leftarrow m$ 
```

Execution of *Euclid*(72,45)

Step#	Action	State#	m	n	r
		0	72	45	
1	$m <> n$				
		1	72	45	
2	$m > n$				
		2	72	45	
3	$m \leftarrow m - n$				
		3	27	45	
4	$m <> n$				
		4	27	45	
5	$m > n$				
		5	27	45	
6	$n \leftarrow n - m$				
		6	27	18	
7	$m <> n$				
		7	27	18	
8	$m > n$				
		8	27	18	
9	$m \leftarrow m - n$				
		9	9	18	
10	$m <> n$				
		10	9	18	
11	$m > n$				
		11	9	18	
12	$n \leftarrow n - m$				
		12	9	9	
13	$m <> n$				
		13	9	9	
14	$r \leftarrow m$				
		14	9	9	9

Correctness

Algorithm $A(\dots)$

Input : In

Output : Out

Method : C

Definition 2.3.1 The algorithm A is *correct* if any process for the transition system $Commands$ starting in a configuration $[C, \sigma]$, where σ satisfies In , is finite and ends with a configuration σ' satisfying Out . □

Decorations

$[\{I\}\text{while } b \text{ do } C; C', \sigma] \triangleright [C; \{I\}\text{while } b \text{ do } C; C', \sigma] \text{ if } \sigma(b) = \text{true}.$

Definition 2.3.2 An assertion U of a decorated algorithm is **valid for a process** if for all configurations of the form $[\{U\}C, \sigma]$ in the process, the assertion U is satisfied by the state σ . □

Euclid's algorithm (decorated)

```
Algorithm Euclid( $m, n$ )  
Input      :  $m, n \geq 1$   
Output     :  $r = \text{gcd}(m_0, n_0)$   
Method     : {  $I$  } while  $m \neq n$  do  
             if  $m > n$  then  
                  $m \leftarrow m - n$   
             else  
                  $n \leftarrow n - m;$   
              $r \leftarrow m$ 
```

$$I : \text{gcd}(m, n) = \text{gcd}(m_0, n_0),$$

Validity

Algorithm $A(\dots)$

Input : In

Output : Out

Method : C

Definition 2.3.3 The algorithm A is *valid* if all its assertions are valid for all processes starting in a configuration $[C, \sigma]$ where σ satisfies In . □

Proof-burdens

$$\{U\}C\{V\}$$

For any state σ , if σ satisfies U and the execution of C in σ leads to σ' (ie. $[C, \sigma] \triangleright^* \sigma'$), then σ' must satisfy V .

Proof principle for simple proof-burdens

Proof principle for simple proof-burdens Let $C = c_1; \dots; c_k$ be a sequence of assignments and let x_1, \dots, x_n be the variables of the algorithm. Suppose that C executed in σ leads to σ' . Then the proof-burden $\{U\}C\{V\}$ is proved by proving the implication

$$U(x_1, \dots, x_n) \Rightarrow V(x'_1, \dots, x'_n)$$

—where x'_1, \dots, x'_n are the values of the variables in σ' expressed as functions of their values in σ .

Proof principle for compound proof-burdens

Proof principle for compound proof-burdens A proof-burden of the form $\{U\}C_1; C_2\{V\}$ gives rise to the proof-burdens

$$\{U\}C_1\{W\} \quad \text{and} \quad \{W\}C_2\{V\}.$$

A proof-burden of the form $\{U\}\text{if } b \text{ then } C_1 \text{ else } C_2\{V\}$ gives rise to the proof-burdens

$$\{U \wedge b\}C_1\{V\} \quad \text{and} \quad \{U \wedge \neg b\}C_2\{V\}.$$

A proof-burden of the form $\{U\}\text{while } b \text{ do } C\{V\}$ gives rise to the proof-burdens

$$\begin{aligned} U &\Rightarrow I && \text{(basis)} \\ \{I \wedge b\}C\{I\} &&& \text{(invariance)} \\ I \wedge \neg b &\Rightarrow V && \text{(conclusion).} \end{aligned}$$

Termination

Algorithm $A(\dots)$

Input : In

Output : Out

Method : C

Definition 2.3.4 We say that A *terminates* if any process starting in a configuration $[C, \sigma]$, where σ satisfies In , is finite. \square

Termination principle

Termination principle for algorithms Let x_1, \dots, x_n be the variables of an algorithm A. A terminates if for every loop

$\{I\}$ while b do C

in its method with I as valid invariant, there exists an integer-valued function $\mu(x_1, \dots, x_n)$ satisfying

a) $I \Rightarrow \mu(x_1, \dots, x_n) \geq 0$

b) $I \wedge b \Rightarrow \mu(x_1, \dots, x_n) > \mu(x'_1, \dots, x'_n) \geq 0$

—where x'_1, \dots, x'_n are the values of the variables after an iteration expressed as functions of their values before.

Extended version of Euclid's algorithm

Algorithm ExtendedEuclid(m, n)
Input : $m, n \geq 1$
Output : $(r = \text{gcd}(m_0, n_0)) \wedge (s = \text{lcm}(m_0, n_0))$
Method : $p \leftarrow m; q \leftarrow n;$
 $\{I\}$ while $m \neq n$ do
 if $m > n$ then
 $m \leftarrow m - n; p \leftarrow p + q$
 else
 $n \leftarrow n - m; q \leftarrow q + p;$
 $r \leftarrow m; s \leftarrow (p + q)/2$

$$I : (mq + np = 2m_0n_0) \wedge (\text{gcd}(m, n) = \text{gcd}(m_0, n_0)).$$

Factorial

```
Algorithm Factorial( $n$ )  
Input      :  $n \geq 0$   
Output     :  $r = n_0!$   
Method     :  $r \leftarrow 1$ ;  
            {  $I$  } while  $n \neq 0$  do  
                 $r \leftarrow r * n$ ;  
                 $n \leftarrow n - 1$ ;
```

$$I : (r = n_0! / n!) \wedge (n_0 \geq n \geq 0).$$

Power sum

Algorithm PowerSum(x, n)
Input : $(x \neq 0) \wedge (n \geq 0)$
Constants: x, n
Output : $r = \sum_{i=0}^n x^i$
Method : $r \leftarrow 1; m \leftarrow 0;$
 { I } **while** $m \neq n$ **do**
 $r \leftarrow r * x + 1;$
 $m \leftarrow m + 1;$

$$I : (r = \sum_{i=0}^m x^i) \wedge (n \geq m \geq 0).$$

Finding maximum in an array

Algorithm ArrayMax(A)

Input : true

Constants: A

Output : $r = \max A$

Method : $r \leftarrow -\infty; i \leftarrow 0;$

$\{I\}$ **while** $i \neq |A|$ **do**

if $r < A[i]$ **then** $r \leftarrow A[i];$

$i \leftarrow i + 1$

$$I : (0 \leq i \leq |A|) \wedge (r = \max A[0..i]) \quad \mu(A, i, r) = |A| - i$$

Time complexity

Algorithm $A(x_1, \dots, x_n)$

Input : In

Output : Out

Method : C

Definition 2.6.1 The *time complexity of A* is the function $T[A]$ taking a state σ satisfying In to the length of the process starting at $[C, \sigma]$. \square

We can regard $T[A]$ as a function of the input parameters x_1, \dots, x_n

Time complexity for ArrayMax

```
 $r \leftarrow -\infty; i \leftarrow 0;$   
 $\{I\}$  while  $i \neq |A|$  do  
    if  $r < A[i]$  then  $r \leftarrow A[i];$   
     $i \leftarrow i + 1$ 
```

Steps

2

$|A| + 1$

between $|A|$ and $2|A|$

$|A|$

Total number of steps:

$$3|A| + 3 \leq T[\text{ArrayMax}](A) \leq 4|A| + 3$$

More precisely, $T[\text{ArrayMax}](A)$ equals $3|A| + 3$ plus the number of times $A[i]$ is strictly larger than $\max A[0..i]$, with i running through the indices $0, \dots, |A| - 1$.

Time complexity for Euclid

```

while  $m \neq n$  do
    if  $m > n$  then
         $m \leftarrow m - n$ 
    else
         $n \leftarrow n - m$ 

```

m	n	T	m	n	T	m	n	T	m	n	T	m	n	T	...
1	1	2	2	1	5	3	1	8	4	1	11	5	1	14	...
1	2	5	2	2	2	3	2	8	4	2	5	5	2	11	...
1	3	8	2	3	8	3	3	2	4	3	11	5	3	11	...
1	4	11	2	4	5	3	4	11	4	4	2	5	4	14	...
1	5	14	2	5	11	3	5	11	4	5	14	5	5	2	...
1	6	17	2	6	8	3	6	5	4	6	8	5	6	17	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Worst-case time complexity

Algorithm $A(x_1, \dots, x_n)$

Input : In

Output : Out

Method : C

Definition 2.6.2 Let $size$ be a function mapping states satisfying In to non-negative integers. The *worst-case time complexity* of A is the function mapping $n \geq 0$ to the maximum of $T[A](\sigma)$ for states σ (satisfying In) with $size(\sigma) = n$. \square

We can regard $size$ as a function of the input parameters x_1, \dots, x_n , and write

$$T[\text{ArrayMax}](A) = 4|A| + 3 \in \mathcal{O}(|A|)$$

Exponentiation in linear time

Algorithm LinExp(x, p)

Input : $p \geq 0$

Constants: x, p

Output : $r = x^p$

Method : $r \leftarrow 1; q \leftarrow p;$

 {I} **while** $q > 0$ **do**

$r \leftarrow r * x; q \leftarrow q - 1$

$$I : (rx^q = x^p) \wedge (q \geq 0) \quad \mu(x, p, r, q) = q$$

Exponentiation in logarithmic time

Algorithm LogExp(x, p)

Input : $p \geq 0$

Constants: x, p

Output : $r = x^p$

Method : $r \leftarrow 1; q \leftarrow p; h \leftarrow x;$

 {I} **while** $q > 0$ **do**

if q even **then**

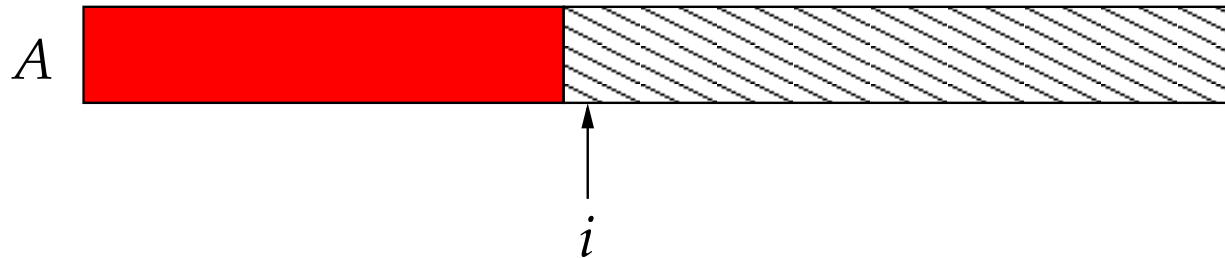
$q \leftarrow q/2; h \leftarrow h * h$

else

$q \leftarrow q - 1; r \leftarrow r * h$

$$I : (rh^q = x^p) \wedge (q \geq 0) \quad \mu(x, p, r, q, h) = q$$

Scanning



$$I : (0 \leq i \leq |A|) \wedge I' \quad \mu(A, i, \dots) = |A| - i$$

```
<< init >>;  $i \leftarrow 0$ ;  
{ $I$ } while  $i \neq |A|$  do  
    << update >>;  $i \leftarrow i + 1$ ;  
<< end >>
```

Linear search

Algorithm LinearSearch(A, s)

Input : true

Constants: A, s

Output : $(0 \leq r \leq |A|) \wedge (s \notin A[0..r]) \wedge (r = |A| \vee A[r] = s)$

Method : $r \leftarrow 0;$

$\{I\}$ **while** $(r \neq |A|) \wedge (A[r] \neq s)$ **do**
 $r \leftarrow r + 1;$

$$I : (0 \leq r \leq |A|) \wedge (s \notin A[0..r]) \quad \mu(A, s, r) = |A| - r$$

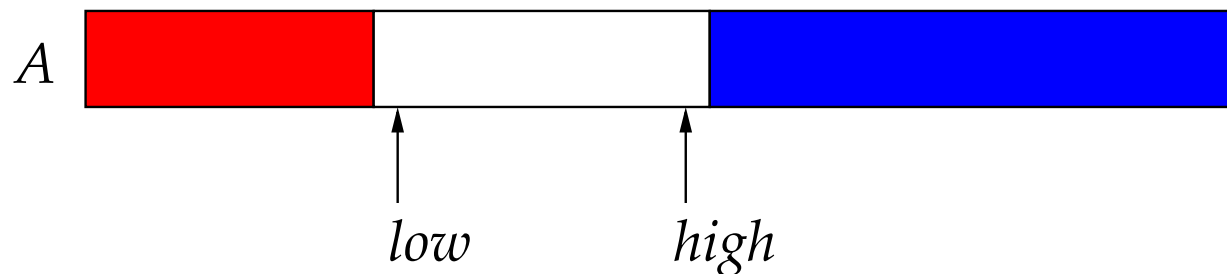
Binary search: specification and invariant

Algorithm BinarySearch(A, s)

Input : A sorted

Constants: A, s

Output : $(0 \leq r \leq |A|) \wedge (A[0..r] < s) \wedge (s \leq A[r..|A|])$



```
 $low \leftarrow 0; high \leftarrow |A|;$   
 $\{I\}$  while  $low \neq high$  do  
     $\ll$  narrow the gap  $\gg;$   
 $r \leftarrow low$ 
```

Binary search: the algorithm

Algorithm BinarySearch(A, s)

Input : A sorted

Constants: A, s

Output : $(0 \leq r \leq |A|) \wedge (A[0..r] < s) \wedge (s \leq A[r..|A|])$

Method : $low \leftarrow 0; high \leftarrow |A|;$

 {I} **while** $low \neq high$ **do**

$m \leftarrow (low + high) / 2;$

if $A[m] < s$ **then**

$low \leftarrow m + 1$

else

$high \leftarrow m;$

$r \leftarrow low$

$I : (0 \leq low \leq high \leq |A|) \wedge (A[0..low] < s) \wedge (s \leq A[high..|A|])$

$\mu(A, s, low, high, m, r) = high - low$

Insertion sort

Algorithm InsertionSort(A)

Input : true

Output : $(A \text{ perm } A_0) \wedge (A \text{ sorted})$

Method : $i \leftarrow 0$;

 { I } **while** $i \neq |A|$ **do**

$j \leftarrow i$;

 { J } **while** $(j \neq 0) \wedge (A[j-1] > A[j])$ **do**

$A[j-1] \leftrightarrow A[j]$;

$j \leftarrow j - 1$;

$i \leftarrow i + 1$

$I : (A \text{ perm } A_0) \wedge (0 \leq i \leq |A|) \wedge (A[0..i] \text{ sorted})$

$J : (A \text{ perm } A_0) \wedge (0 \leq j \leq i < |A|) \wedge (A[0..j], A[j..i+1] \text{ sorted})$

$$\mu_I(A, i, j) = |A| - i \quad \mu_J(A, i, j) = j$$

Merge sort

Algorithm MergeSort(A)

Input : true

Output : $(A \text{ perm } A_0) \wedge (A \text{ sorted})$

Method : **if** $|A| > 1$ **then**
 $B \leftarrow A[0..|A|/2];$
 $C \leftarrow A[|A|/2..|A|];$
 MergeSort(B);
 MergeSort(C);
 { W } Merge(A, B, C)

Algorithm Merge(A, B, C)

Input : $(|A| = |B| + |C|) \wedge (B, C \text{ sorted})$

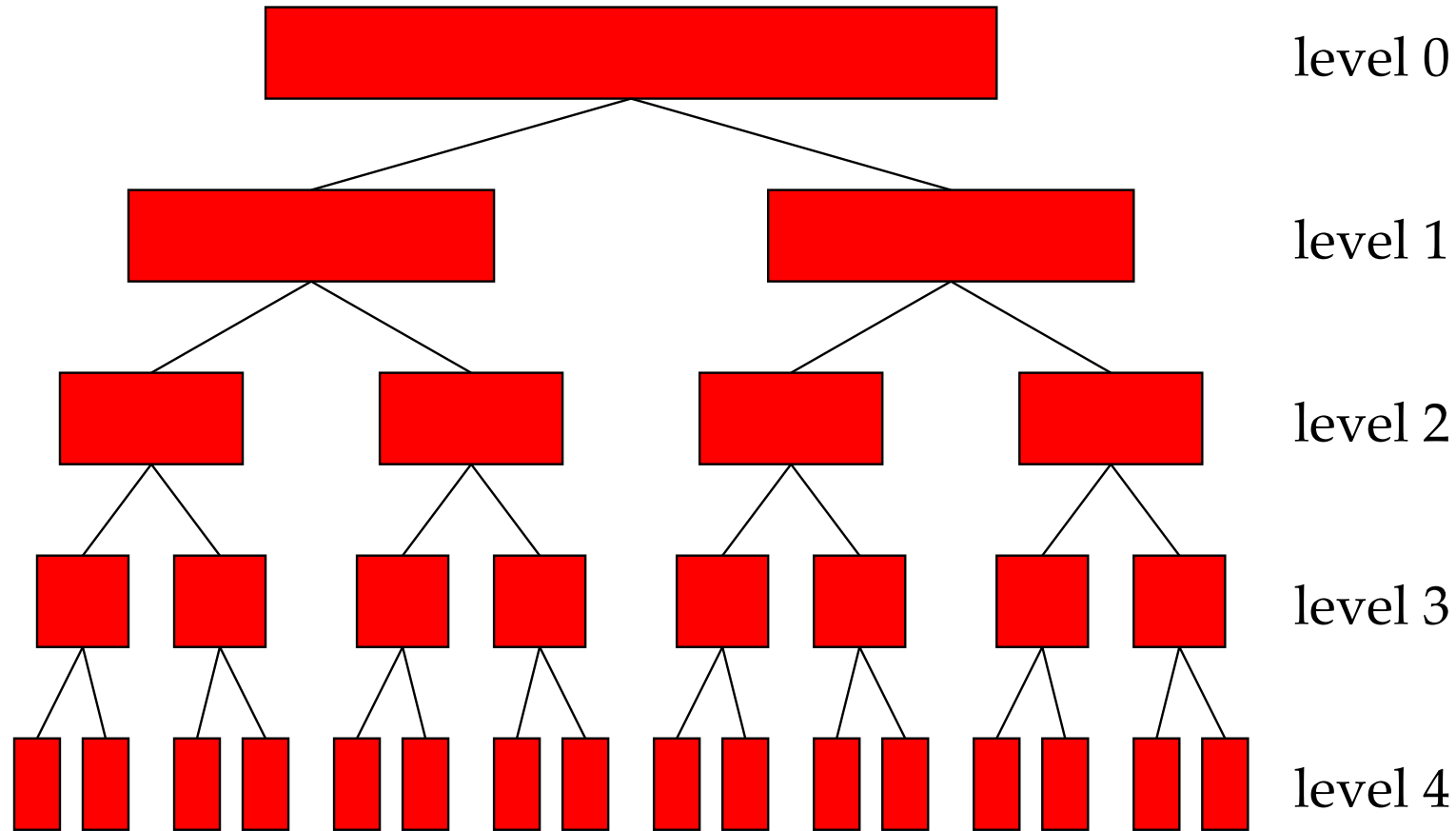
Constants: B, C

Output : $(A \text{ perm } BC) \wedge (A \text{ sorted})$

$W : (A = A_0) \wedge (B \text{ perm } A_1) \wedge (C \text{ perm } A_2) \wedge (B, C \text{ sorted})$

$\mu(A) = |A|$

Merge sort: time complexity, assuming n a power of 2



Time spent at level i (for $0 \leq i \leq \log n$): $2^i \cdot n/2^i = n$ time units.

Merge: specification

Algorithm Merge(A, B, C)

Input : $(|A| = |B| + |C|) \wedge (B, C \text{ sorted})$

Constants: B, C

Output : $(A \text{ perm } BC) \wedge (A \text{ sorted})$

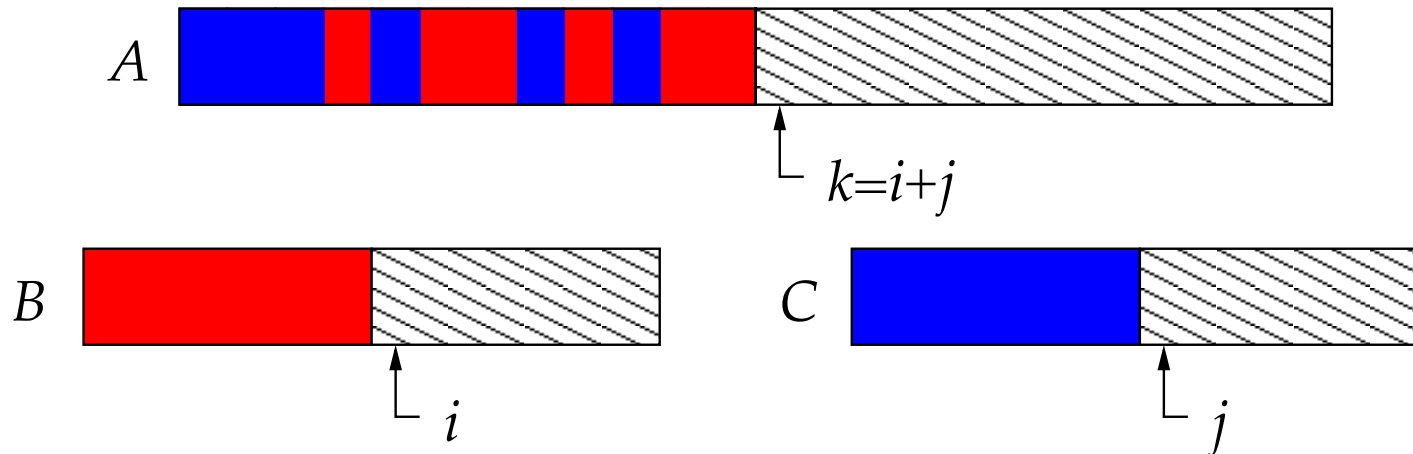
So, with $|A| = 11$ and

$$B = [1, 1, 4, 7, 8] \quad \text{and} \quad C = [1, 2, 2, 3, 4, 7]$$

we should obtain

$$A = [1, 1, 1, 2, 2, 3, 4, 4, 7, 7, 8].$$

Merge: scanning invariant



$I : (|A| = |B| + |C|) \wedge (0 \leq i \leq |B|) \wedge (0 \leq j \leq |C|) \wedge (A[0..i+j] \text{ perm } B[0..i]C[0..j]) \wedge (A[0..i+j] \text{ sorted})$

```
<< init >>;  
{I} while  $i + j \neq |A|$  do  
    << update >>;  
<< end >>
```

Merge: the algorithm

Algorithm Merge(A, B, C)

Input : $(|A| = |B| + |C|) \wedge (B, C \text{ sorted})$

Constants: B, C

Output : $(A \text{ perm } BC) \wedge (A \text{ sorted})$

Method : $i \leftarrow 0; j \leftarrow 0;$

$\{I\}$ **while** $i + j \neq |A|$ **do**

if $(i < |B|) \wedge (j = |C| \vee B[i] \leq C[j])$ **then**

$A[i + j] \leftarrow B[i]; i \leftarrow i + 1$

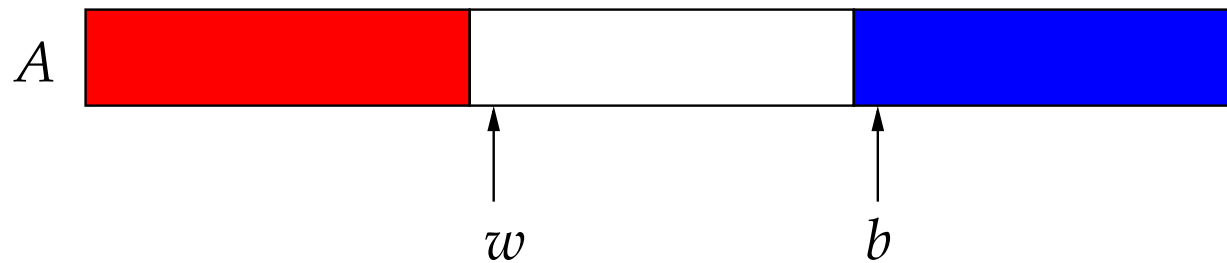
else

$A[i + j] \leftarrow C[j]; j \leftarrow j + 1$

$I : (|A| = |B| + |C|) \wedge (0 \leq i \leq |B|) \wedge (0 \leq j \leq |C|) \wedge$
 $(A[0..i + j] \text{ perm } B[0..i]C[0..j]) \wedge (A[0..i + j] \text{ sorted})$

$\mu(A, B, C, i, j) = |A| - (i + j)$

Dutch flag: specification



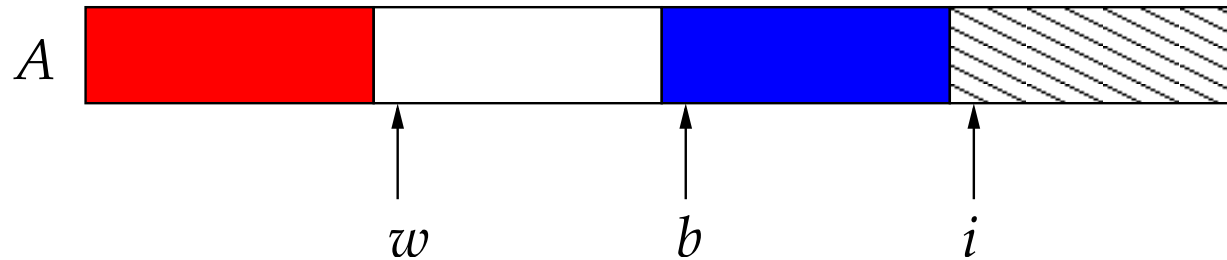
Algorithm DutchFlag(A, s)

Input : true

Constants: s

Output : $(A \text{ perm } A_0) \wedge (0 \leq w \leq b \leq |A|) \wedge$
 $(A[0..w] < s) \wedge (A[w..b] = s) \wedge (A[b..|A|] > s)$

Dutch flag: scanning invariant

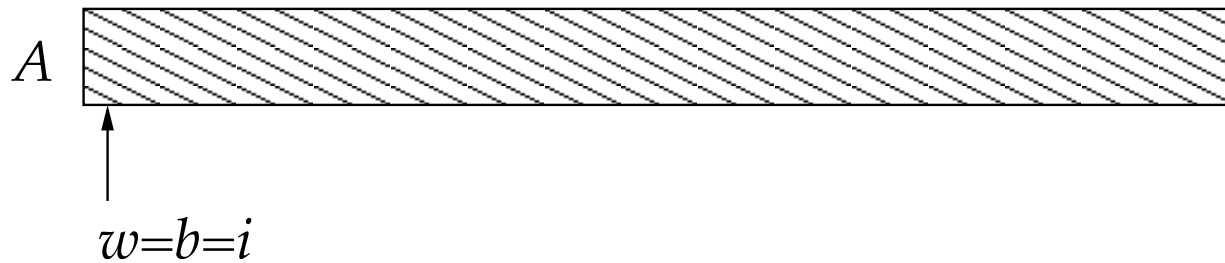


$$I : (A \text{ perm } A_0) \wedge (0 \leq w \leq b \leq i \leq |A|) \wedge \\ (A[0..w] < s) \wedge (A[w..b] = s) \wedge (A[b..i] > s)$$

```
<< init >>;  $i \leftarrow 0$ ;  
{ $I$ } while  $i \neq |A|$  do  
    << update >>;  $i \leftarrow i + 1$ ;  
<< end >>
```

Dutch flag: basis

$\{\text{true}\} \ll \text{init} \gg; i \leftarrow 0\{I\}$



$\ll \text{init} \gg = w \leftarrow 0; b \leftarrow 0$

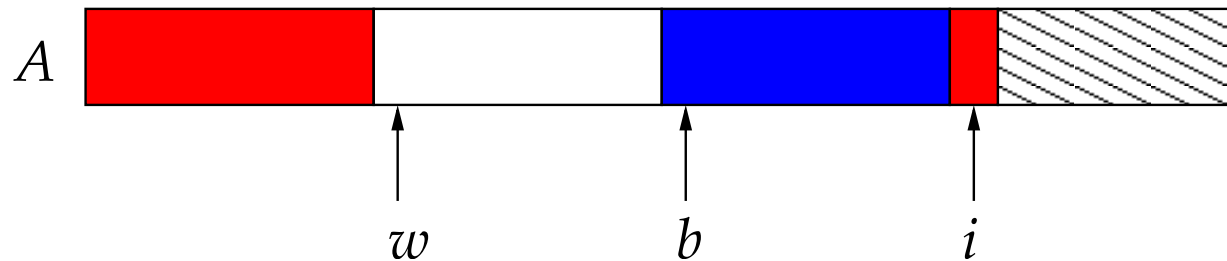
Dutch flag: invariance

$$\{I \wedge i \neq |A|\} \ll \text{update} \gg; i \leftarrow i + 1 \{I\}$$
$$\begin{aligned} \ll \text{update} \gg = & \text{if } A[i] < s \text{ then} \\ & \ll \text{red} \gg \\ & \text{else if } A[i] = s \text{ then} \\ & \ll \text{white} \gg \\ & \text{else} \\ & \ll \text{blue} \gg \end{aligned}$$

- 1 : $\{I \wedge (i \neq |A|) \wedge (A[i] < s)\} \ll \text{red} \gg; i \leftarrow i + 1 \{I\}$
- 2 : $\{I \wedge (i \neq |A|) \wedge (A[i] = s)\} \ll \text{white} \gg; i \leftarrow i + 1 \{I\}$
- 3 : $\{I \wedge (i \neq |A|) \wedge (A[i] > s)\} \ll \text{blue} \gg; i \leftarrow i + 1 \{I\}$

Dutch flag: invariance, red

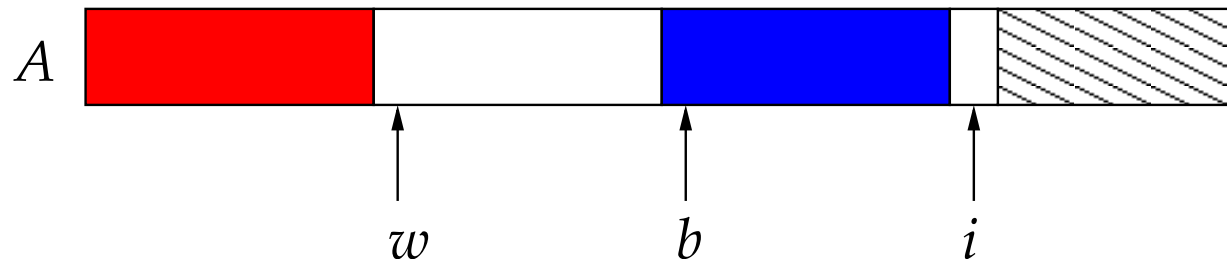
$$\{I \wedge (i \neq |A|) \wedge (A[i] < s)\} \ll \text{red} \gg; i \leftarrow i + 1 \{I\}$$



$$\ll \text{red} \gg = A[i] \leftrightarrow A[w]; w \leftarrow w + 1; A[i] \leftrightarrow A[b]; b \leftarrow b + 1$$

Dutch flag: invariance, white

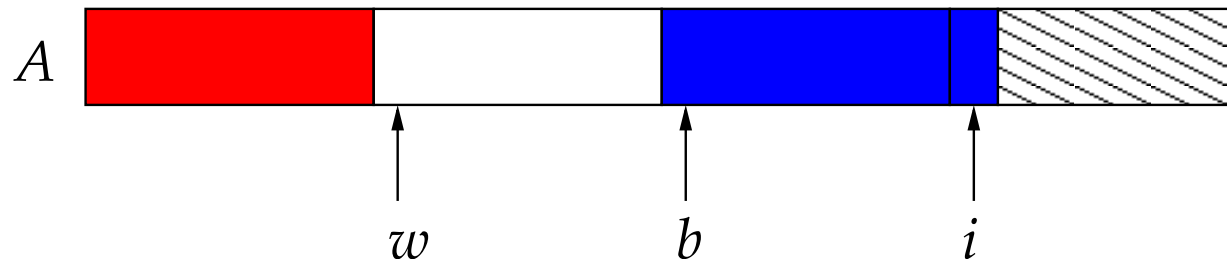
$$\{I \wedge (i \neq |A|) \wedge (A[i] = s)\} \ll \text{white} \gg; i \leftarrow i + 1 \{I\}$$



$$\ll \text{white} \gg = A[i] \leftrightarrow A[b]; b \leftarrow b + 1$$

Dutch flag: invariance, blue

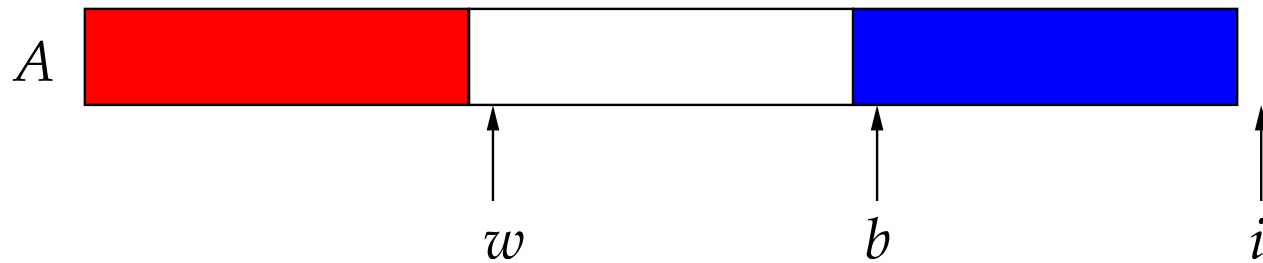
$$\{I \wedge (i \neq |A|) \wedge (A[i] > s)\} \ll \text{blue} \gg; i \leftarrow i + 1 \{I\}$$



$$\ll \text{blue} \gg = \lambda$$

Dutch flag: conclusion

$$\{I \wedge (i = |A|)\} \ll \text{end} \gg \{Out\}$$



$$\ll \text{end} \gg = \lambda$$

Dutch flag: the algorithm

Algorithm DutchFlag(A, s)

Input : true

Constants: s

Output : $(A \text{ perm } A_0) \wedge (0 \leq w \leq b \leq |A|) \wedge$
 $(A[0..w] < s) \wedge (A[w..b] = s) \wedge (A[b..|A|] > s)$

Method : $w \leftarrow 0; b \leftarrow 0; i \leftarrow 0;$

$\{I\}$ **while** $i \neq |A|$ **do**

if $A[i] < s$ **then**

$A[i] \leftrightarrow A[w]; w \leftarrow w + 1; A[i] \leftrightarrow A[b]; b \leftarrow b + 1$

else if $A[i] = s$ **then**

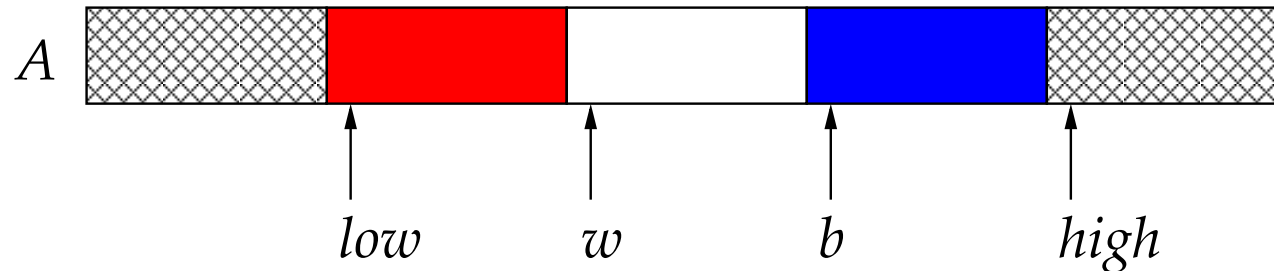
$A[i] \leftrightarrow A[b]; b \leftarrow b + 1;$

$i \leftarrow i + 1$

$I : (A \text{ perm } A_0) \wedge (0 \leq w \leq b \leq i \leq |A|) \wedge$
 $(A[0..w] < s) \wedge (A[w..b] = s) \wedge (A[b..i] > s)$

$$\mu(A, s, w, b, i) = |A| - i$$

Dutch flag: from *low* to *high*



Algorithm DutchFlag($A, low, high, s$)

Input : $0 \leq low \leq high \leq |A|$

Constants: $low, high, A[0..low], A[high..|A|], s$

Output : $(A[low..high] \text{ perm } A_0[low..high]) \wedge$

$(low \leq w \leq b \leq high) \wedge$

$(A[low..w] < s) \wedge (A[w..b] = s) \wedge (A[b..high] > s)$

Quick sort

Algorithm QuickSort($A, low, high$)

Input : $0 \leq low \leq high \leq |A|$

Constants: $low, high, A[0..low], A[high..|A|]$

Output : $(A[low..high] \text{ perm } A_0[low..high]) \wedge (A[low..high] \text{ sorted})$

Method : **if** $high - low > 1$ **then**

$r \leftarrow \text{random}(low, high);$

$(w, b) \leftarrow \text{DutchFlag}(A, low, high, A[r]);$

QuickSort(A, low, w);

QuickSort($A, b, high$)

$$\mu(A, low, high) = high - low$$

Quick select

```
Algorithm QuickSelect(A, low, high, k)  
Input      : ( $0 \leq low \leq k < high \leq |A|$ )  
Constants: low, high, A[0..low], A[high..|A|], k  
Output     : (A[low..high] perm A0[low..high])  $\wedge$   
            (A[low..k]  $\leq$  A[k]  $\leq$  A[k..high])  
Method     :  $r \leftarrow \text{random}(low, high)$ ;  
            ( $w, b$ )  $\leftarrow$  DutchFlag(A, low, high, A[r]);  
            if  $k < w$  then  
                QuickSelect(A, low, w, k)  
            else if  $k \geq b$  then  
                QuickSelect(A, b, high, k)
```

$$\mu(A, low, high) = high - low$$

Maximal subsum

Algorithm MaxSubsum(A)

Input : true

Constants: A

Output : $r = \text{ms}(A)$

Method : $\llcorner \text{init} \lrcorner; i \leftarrow 0;$
 $\{I\}$ **while** $i \neq |A|$ **do**
 $\llcorner \text{update} \lrcorner; i \leftarrow i + 1;$
 $\llcorner \text{end} \lrcorner$

$$I : (0 \leq i \leq |A|) \wedge (r = \text{ms}(A[0..i])) \wedge \dots$$

Maximal subsum: maintaining the invariant

$\text{ms}(A[0..i + 1])$ is the maximum of

the maximal subsum obtained by not using $A[i]$, and
the maximal subsum obtained by using $A[i]$

So, we remember also the *maximal right subsum* $\text{mrs}(A[0..i])$

$$I : (0 \leq i \leq |A|) \wedge (r = \text{ms}(A[0..i])) \wedge (h = \text{mrs}(A[0..i]))$$

Maximal subsum: the algorithm

```
Algorithm MaxSubsum( $A$ )  
Input      : true  
Constants:  $A$   
Output     :  $m = \text{ms}(A)$   
Method     :  $r \leftarrow 0; h \leftarrow 0; i \leftarrow 0;$   
             { $I$ } while  $i \neq |A|$  do  
                $h \leftarrow \max\{h + A[i], 0\};$   
                $r \leftarrow \max\{r, h\};$   
                $i \leftarrow i + 1$ 
```

$$\mu(A, r, h, i) = |A| - i$$

Longest monotone sequence

Algorithm LLMS(A)

Input : true

Constants: A

Output : $r = \text{llms}(A)$

Method : $\ll \text{init} \gg; i \leftarrow 0;$
 $\{I\} \mathbf{while} \ i \neq |A| \ \mathbf{do}$
 $\ll \text{update} \gg; i \leftarrow i + 1;$
 $\ll \text{end} \gg$

$$I : (0 \leq i \leq |A|) \wedge (r = \text{llms}(A[0..i])) \wedge \dots$$

Longest monotone sequence: maintaining the invariant

In the i 'th iteration of the loop, $B[l]$ for $1 \leq l \leq |A|$ must contain the necessary information about a monotone sequence from $A[0..i]$ of length l whose last element is minimal (if there is no sequence of length l , we just record that).

$$I : (0 \leq i \leq |A|) \wedge (r = \text{llms}(A[0..i])) \wedge \\ (B[l] = \text{ms}_l(A[0..i]) \text{ for } 0 \leq l \leq |A|)$$

$$\text{ms}_l(A[0..i]) = \begin{cases} -\infty & \text{if } l = 0 \\ \text{minimal last element in} \\ \text{a monotone sequence of} & \text{if such exists} \\ \text{length } l \text{ from } A[0..i] & \\ \infty & \text{otherwise} \end{cases}$$

Longest monotone sequence: the algorithm

Algorithm LLMS(A)

Input : true

Constants: A

Output : $r = \text{llms}(A)$

Method : $B \leftarrow [|A| + 1 : \infty]$; $B[0] \leftarrow -\infty$; $r \leftarrow 0$; $i \leftarrow 0$;

$\{I\}$ **while** $i \neq |A|$ **do**

$l \leftarrow \text{BinarySearch}'(B, A[i]);$

$B[l] \leftarrow A[i]$; $r \leftarrow \max\{r, l\}$; $i \leftarrow i + 1$

$$\mu(A, B, r, i) = |A| - i$$

Algorithm BinarySearch'(B, s)

Input : B sorted

Constants: B, s

Output : $(0 \leq l \leq |B|) \wedge (B[0..l] \leq s) \wedge (s < B[l..|B|])$