Dynamic programming

- It has nothing to do with “programming languages”

- Problem: Input $w_1\ w_2 \dots w_n$, t   each $0 \le w_i \le k$
- Output: Number of inputs $x \in \{0,1\}^n : \sum w_i x_i = t$

- Let's try a recursive approach...

- Problem: Input w_1 w_2 … w_n , t   each $0 \leq w_i \leq k$
- Output: Number of inputs $x \in \{0,1\}^n : \sum w_i x_i = t$

- Structure of solutions: $S(i,s) = S(i\text{-}1,s) + S(i\text{-}1,s–w_i)$ i = n,...
- $S(i,s)$ = Number of inputs $x \in \{0,1\}^i : \sum w_i x_i = s$
- Gives recursive approach: T(n) = ?

- Problem: Input $w_1\ w_2 \ldots w_n$, t   each $0 \le w_i \le k$
- Output: Number of inputs $x \in \{0,1\}^n : \sum w_i x_i = t$

- Structure of solutions: $S(i,s) = S(i-1,s) + S(i-1,s\text{–}w_i\ )\ i = n,...$
- $S(i,s)$ = Number of inputs $x \in \{0,1\}^i : \sum w_i x_i = s$
- Gives recursive approach: $T(n) = 2\ T(n-1) \Rightarrow T(n) \ge 2^n$

- How can we do faster when $k \approx n$?

- ?

- Problem: Input $w_1\ w_2 \dots w_n$, t each $0 \leq w_i \leq k$
- Output: Number of inputs $x \in \{0,1\}^n : \sum w_i x_i = t$

- Structure of solutions: $S(i,s) = S(i-1,s) + S(i-1,s–w_i\ )$ i = n,...
- $S(i,s)$ = Number of inputs $x \in \{0,1\}^i : \sum w_i x_i = s$
- Gives recursive approach: $T(n) = 2\ T(n-1) \Rightarrow T(n) \geq 2^n$

- How can we do faster when $k \approx n$?

- Stop solving the same problems over and over again!

- Total sum is $\leq kn$, so there really are only ? different $S(i,t)$

- Problem: Input $w_1\ w_2 \ldots w_n$, t   each $0 \leq w_i \leq k$
- Output: Number of inputs $x \in \{0,1\}^n : \sum w_i\, x_i = t$

- Structure of solutions: $S(i,s) = S(i\text{-}1,s) + S(i\text{-}1,s\text{–}w_i\ )$ i = n,...
- $S(i,s)$ = Number of inputs $x \in \{0,1\}^i : \sum w_i\, x_i = s$
- Gives recursive approach: $T(n) = 2\ T(n\text{-}1) \Rightarrow T(n) \geq 2^n$

- How can we do faster when $k \approx n$?

- Stop solving the same problems over and over again!

- Total sum is $\leq$ kn, so there really are only kn^2 different S(i,t)
  Just solve all of those!

- Problem: Input w_1 w_2 … w_n , t each $0 \le w_i \le k$
- Output: Number of inputs $x \in \{0,1\}^n : \sum w_i x_i = t$

Sum s

	1				
			,,,		
1	2				
1	1				

i = 1...n

- Fill first column
- (for i = 2... n)
  (for s = 0 … kn)
    ?

Algorithm

- Problem: Input $w_1 \; w_2 \; \ldots \; w_n$, t   each $0 \leq w_i \leq k$
- Output: Number of inputs $x \in \{0,1\}^n : \sum w_i x_i = t$

Sum s

	1				
			,,,		
1	2				
1	1				

i = 1...n

- Fill first column
- (for i = 2... n)
  - (for s = 0 … kn)
    - $S(i,s) = S(i-1,s) + S(i-1,s-w_i)$

Algorithm

- T(n) = ?

- Problem: Input $w_1\ w_2 \ldots w_n$, t   each $0 \le w_i \le k$
- Output: Number of inputs $x \in \{0,1\}^n : \sum w_i\, x_i = t$

Sum s

	1				
			,,,		
1	2				
1	1				

i = 1...n

- Fill first column
- (for i = 2... n)

```
(for i = 2... n)
  (for s = 0 … kn)
    $S(i,s) = S(i-1,s) + S(i-1,s-w_i)$
```

- $T(n) = O(kn^2)$

- Problem: Have t and ∞ piles of coins of values d_1 ,... ,d_k
- Want to use minimum number of coins to obtain target t
- Example: d = (5,4,1) t = 8 t = 5+1+1+1, t = 4+4

- Structure of solution: Cost[c] = ?

Suppose you obtain c with some optimal number of coins.

If coin of type d_i was used, then your coins without d_i must be optimal for $c - d_i$

- Problem: Have t and ∞ piles of coins of values d_1 ,... ,d_k
- Want to use minimum number of coins to obtain target t
- Example: d = (5,4,1) t = 8 t = 5+1+1+1, t = 4+4

- Structure of solution: $\mathrm{Cost}[c] = 1 + \min_{i \leq k} \mathrm{Cost}[c - d_i]$

Algorithm:

- Initialize vector Cost to 0
- For (c = 1..t) ???

- Problem: Have t and ∞ piles of coins of values d_1 ,... ,d_k
- Want to use minimum number of coins to obtain target t
- Example: d = (5,4,1) t = 8 t = 5+1+1+1, t = 4+4

- Structure of solution: $\text{Cost}[c] = 1 + \min_{i \leq k} \text{Cost}[c - d_i]$

Algorithm:

- Initialize vector Cost to 0
- For (c = 1..t) $\text{Cost}[c] = 1 + \min_{i \leq k} \text{Cost}[c- d_i]$

- T(n) = kn

- To know which coins to use, store ∀ i what latest coin used. Then can reconstruct solution moving backwards

- Longest common subsequence

- Given two strings X, Y , want to find a longest subsequence Z,

  i.e. string Z whose symbols appear in X, Y in the same order, but not necessarily consecutively

- Example: X = XMJYAUZ

  Y = MZJAWXU

  Z = MJAU

- Definition: For a string $X = (x_1 , x_2 , \ldots , x_n)$, we denote by X_i the prefix $(x_1 , x_2 , \ldots, x_i)$.
- X_0 is the empty subsequence Ø
- Do not confuse x with X
- $LCS(X_i , Y_j) =$ longest subsequence of X_i and Y_j
- Optimal substructure? What if X_i and Y_j end with the same symbol $x_i = y_j$?

- Definition: For a string $X = (x_1 , x_2 , \dots , x_n)$,

  we denote by X_i the prefix $(x_1 , x_2 , \dots, x_i)$.

- X_0 is the empty subsequence Ø

- Do not confuse x with X

- $LCS(X_i , Y_j) =$ longest subsequence of X_i and Y_j

$$LCS\left(X_i, Y_j\right) = \begin{cases} \emptyset & \text{if } \; i = 0 \text{ or } j = 0 \\ \left(LCS\left(X_{i-1}, Y_{j-1}\right), x_i\right) & \text{if } x_i = y_j \\ \text{longest}\left(LCS\left(X_i, Y_{j-1}\right), LCS\left(X_{i-1}, Y_j\right)\right) & \text{if } x_i \ne y_j \end{cases}$$

•http://en.wikipedia.org/wiki/Longest_common_subsequence_problem

```
function LCSLength(X[1..m], Y[1..n])
    C = array(0..m, 0..n)
    for i := 0..m
      C[i,0] = 0
    for j := 0..n
      C[0,j] = 0
    for i := 1..m
      for j := 1..n
        if X[i] = Y[j]
          C[i,j] := C[i-1,j-1] + 1
        else
          C[i,j] := max(C[i,j-1], C[i-1,j])
    return C[m,n]
```

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	0	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	1	1	1	1	1	1	1
3	J	0	1	1	2	2	2	2	2
4	Y	0	1	1	2	2	2	2	2
5	A	0	1	1	2	3	3	3	3
6	U	0	1	1	2	3	3	3	4
7	Z	0	1	2	2	3	3	3	4

$T(m,n) = O(m\ n)$

- As before, if we want to output the sequence, we record which rule was used at each point

$$LCS\left(X_i, Y_j\right) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ \left(LCS\left(X_{i-1}, Y_{j-1}\right), x_i\right) & \text{if } x_i = y_j \\ \text{longest}\left(LCS\left(X_i, Y_{j-1}\right), LCS\left(X_{i-1}, Y_j\right)\right) & \text{if } x_i \neq y_j \end{cases}$$

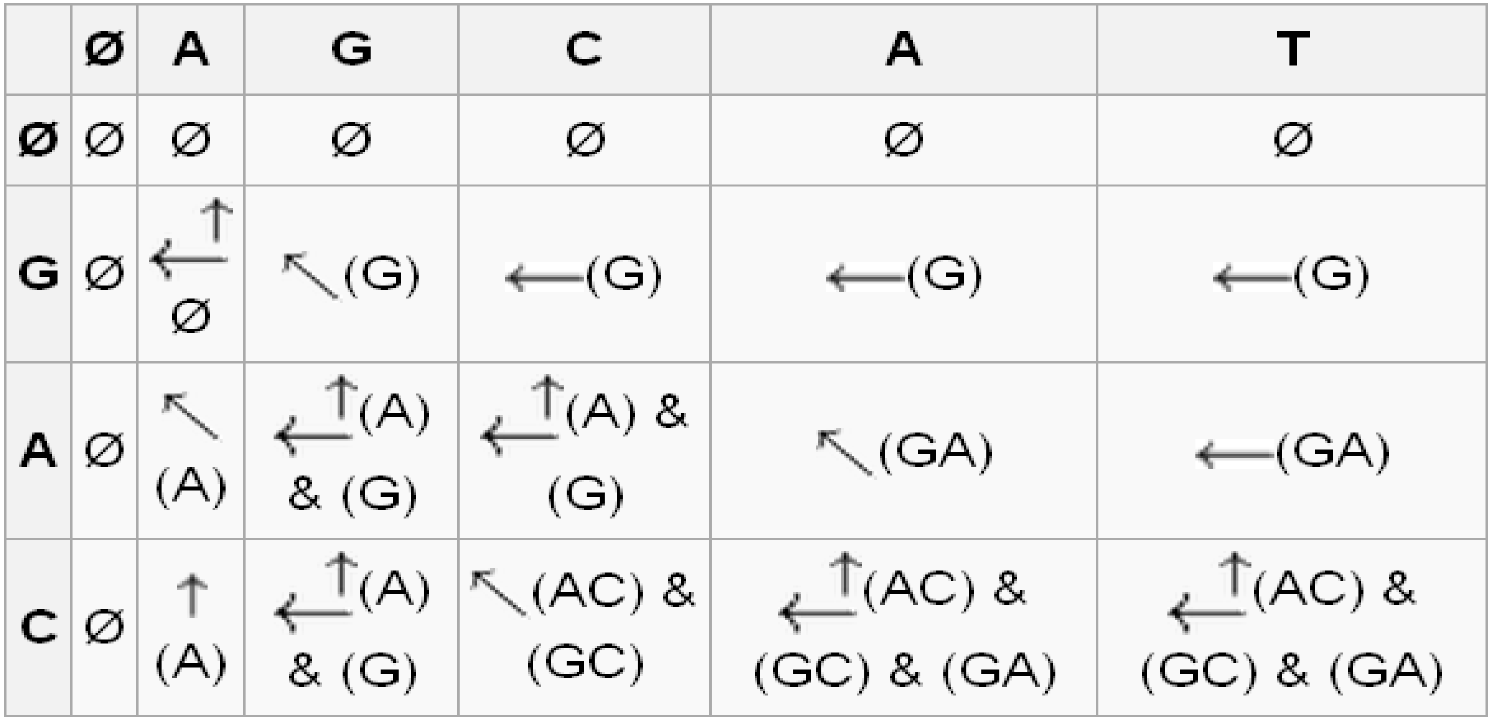
What arrows?

	∅	**A**	**G**	**C**	**A**	**T**
∅	∅	∅	∅	∅	∅	∅
G	∅	↑ ← ∅	↖(G)	←(G)	←(G)	←(G)
A	∅	↖ (A)	←↑(A) & (G)	←↑(A) & (G)	↖(GA)	←(GA)
C	∅	↑ (A)	←↑(A) & (G)	↖(AC) & (GC)	←↑(AC) & (GC) & (GA)	←↑(AC) & (GC) & (GA)

- Then we can reconstruct the sequence backwards.

- As before, if we want to output the sequence, we record which rule was used at each point

$$LCS\left(X_i,Y_j\right)=\begin{cases}\emptyset & \text{if } i=0 \text{ or } j=0\\ \left(LCS\left(X_{i-1},Y_{j-1}\right),x_i\right) & \text{if } x_i=y_j \quad \nwarrow\\ \text{longest}\left(LCS\left(X_i,Y_{j-1}\right),LCS\left(X_{i-1},Y_j\right)\right) & \text{if } x_i\neq y_j \quad \leftarrow \; \uparrow\end{cases}$$



	Ø	**A**	**G**	**C**	**A**	**T**
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	←↑ Ø	↖(G)	←(G)	←(G)	←(G)
A	Ø	↖ (A)	←↑(A) & (G)	←↑(A) & (G)	↖(GA)	←(GA)
C	Ø	↑ (A)	←↑(A) & (G)	↖(AC) & (GC)	←↑(AC) & (GC) & (GA)	←↑(AC) & (GC) & (GA)

- Then we can reconstruct the sequence backwards.

- We have described dynamic programming in an interative “bottom-up” fashion, i.e., solve all the problems from the smallest to the biggest.

- Alternatively, dinamyc programming may be implemented in a “top-down” recursive fashion. You keep a list of the subproblems solved, and at the beginning you check if the current subproblem was already solved, if so you just read off the solution and return.

- This is called Memoization

- Recall even divide-and-conquer may be implemented either in a recursive “top-down” fashion, or in an iterative “bottom-up” fashion.

- Dynamic programming works when we have “Optimal substructure” and “few subproblems”

- Requires solving all subproblems, leads to algorithms with running time usually n^2 or n^3

- Sometimes, greedy is faster.

Greedy Algorithms

A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimized decision in the hope that this choice will lead to a globally optimal solution.

Activity Selection problem

Input: Set of n activities that need the same resource.

$A := \{ a_1, a_2, \ldots\ a_n \}$, $\forall\, i \in [n]$ $a_i = [s_i , f_i)$.

Activity a_i takes time $[s_i , f_i)$. Activities a_i , a_j are compatible if $s_j \geq f_i$.

Activity Selection problem

Input: Set of n activities that need the same resource.

$A := \{ a_1, a_2, \dots a_n \}$, $\forall i \in [n] \;\; a_i = [s_i , f_i)$.

Activity a_i takes time $[s_i , f_i)$. Activities a_i , a_j are compatible if $s_j \geq f_i$.

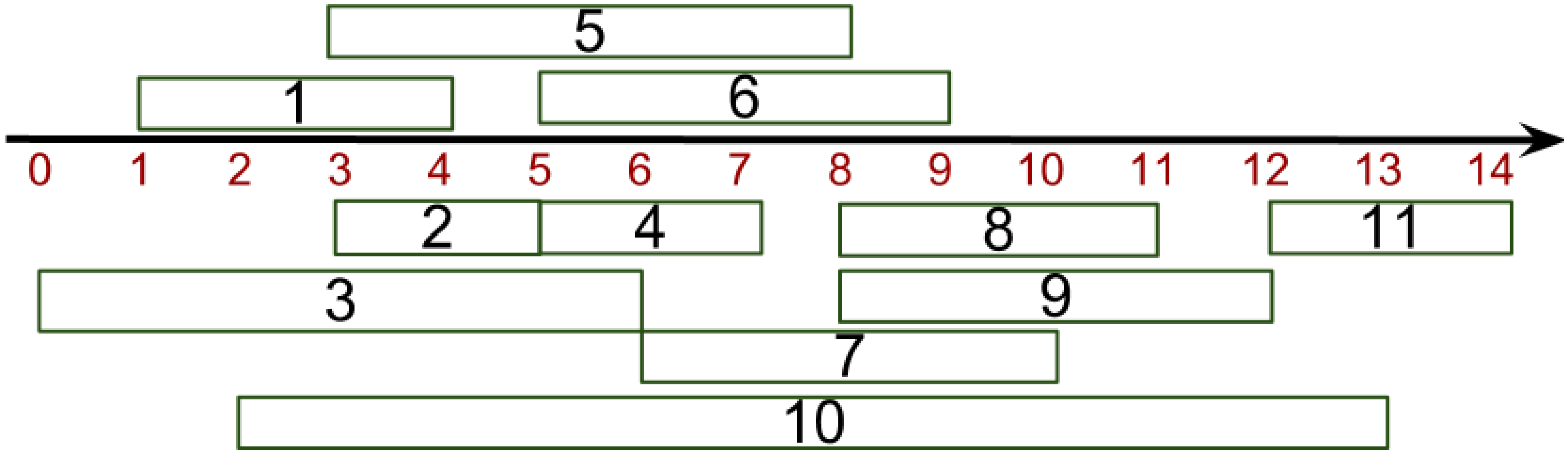
Output:

Maximum size subset of mutually compatible activities.

Example:

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

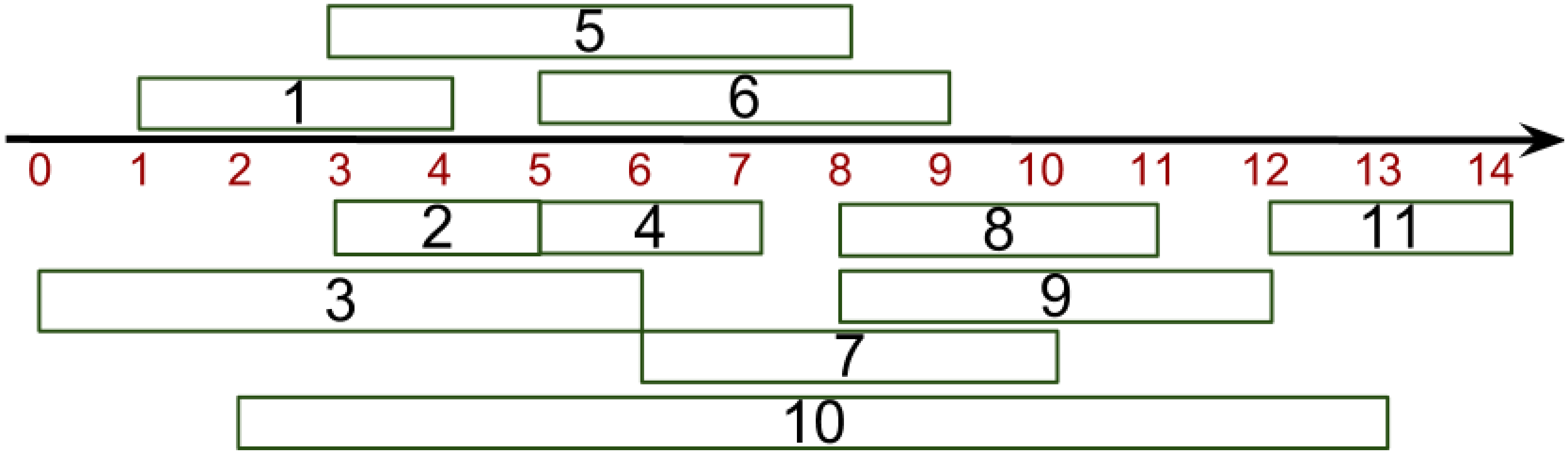
A set of compatible activities = ?



Example:

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

A set of compatible activities = (a_3, a_9, a_{11}).

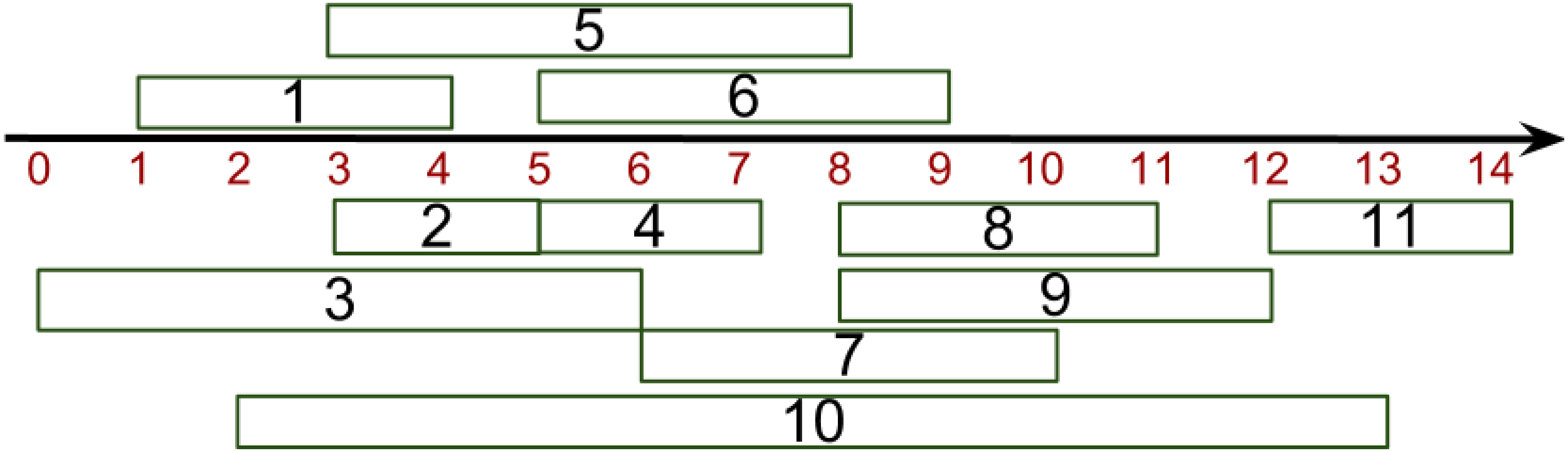


Example:

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

A set of compatible activities = (a_3,a_9,a_{11}).

A maximal set of compatible activities = ?

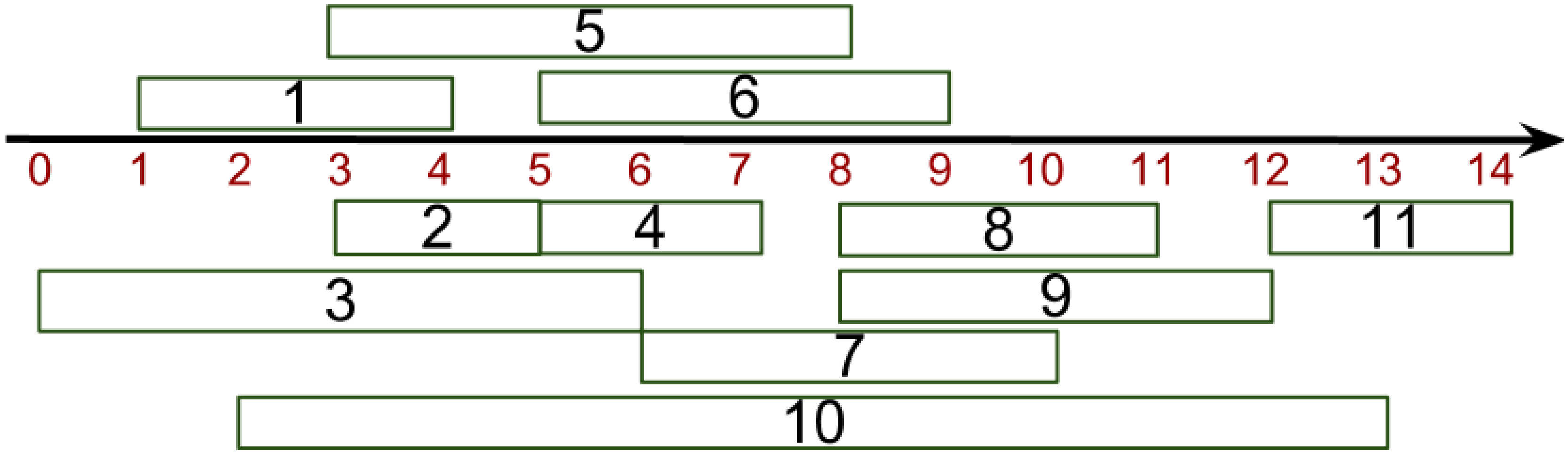


Example:

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

A set of compatible activities = (a_3, a_9, a_{11}).

A maximal set of compatible activities = (a_1, a_4, a_8, a_{11}).



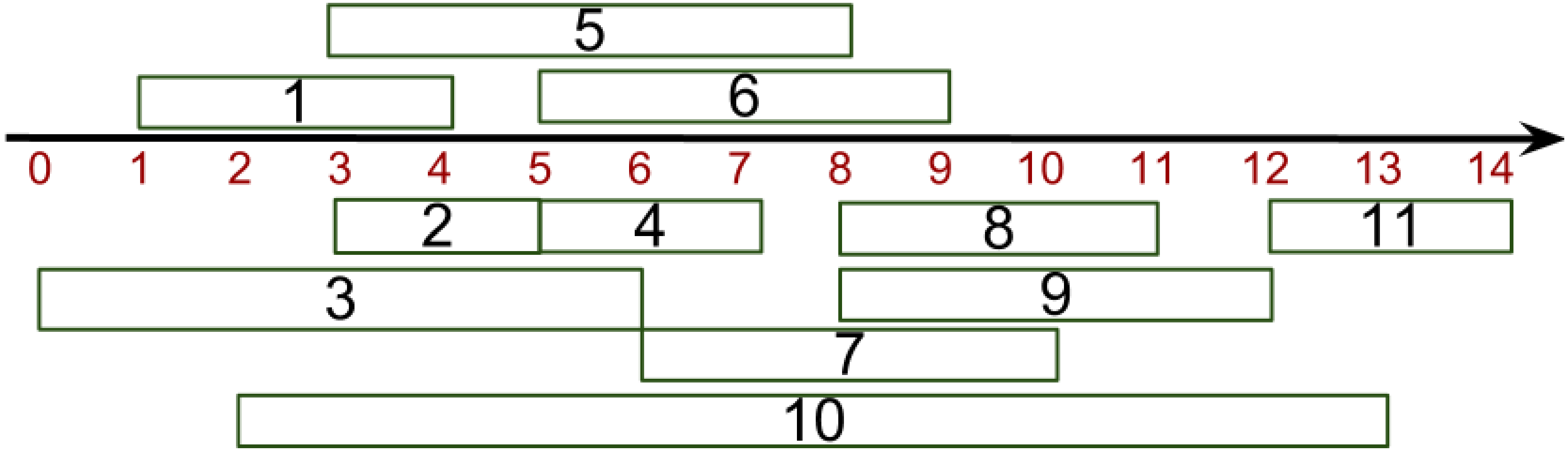
Example:

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

A set of compatible activities = (a_3, a_9, a_{11}).

A maximal set of compatible activities = (a_1, a_4, a_8, a_{11}).

Is there another maximal set ?



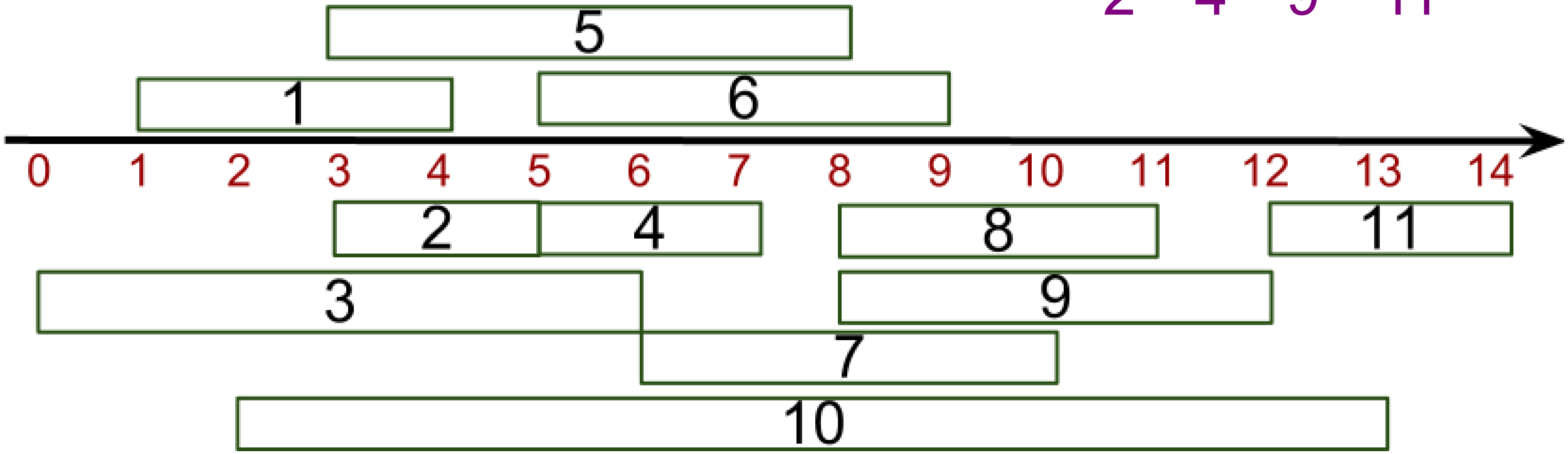
Example:

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

A set of compatible activities = (a_3,a_9,a_{11}).

A maximal set of compatible activities = (a_1,a_4,a_8,a_{11}).

Is there another maximal set? Yes. (a_2,a_4,a_9,a_{11})



- Claim: some optimal solution contains activity with earliest finish time

- Greedy Algorithm:
  Pick activity with earliest finish time,
    that does not overlap with activities already picked
  Repeat

- Let us see it in more detail

Greedy activity selection algorithm

```
activity-selection(A) {
sort A increasingly according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;
    }
return S
}
```

```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
         Add a[i] to S;
         i :=m;} return S;
}
```

Example:

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

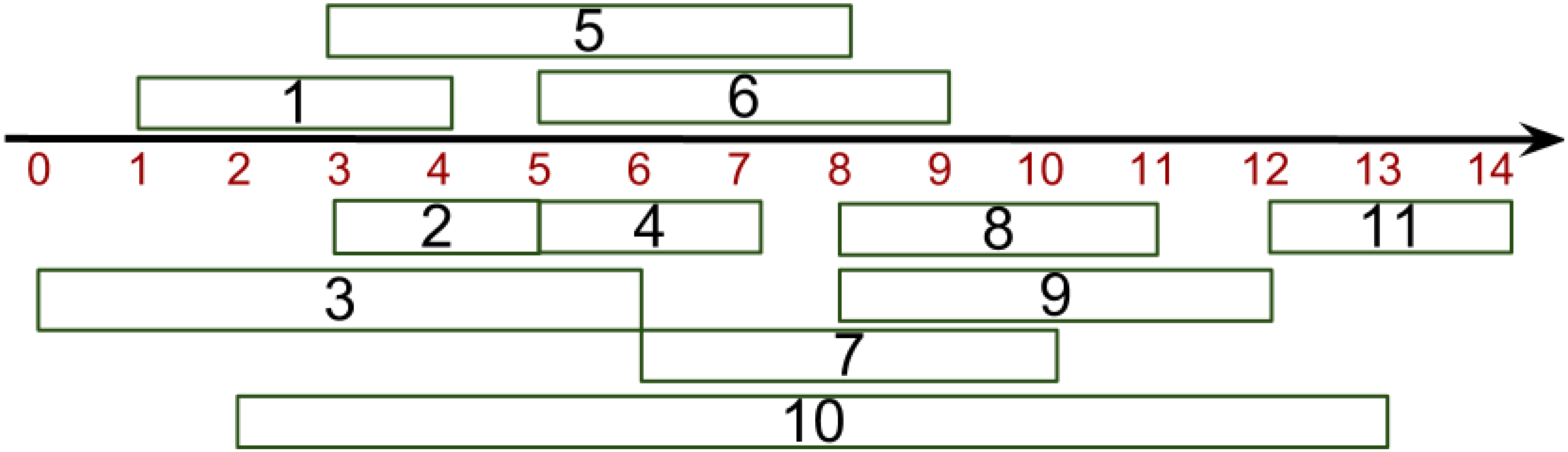
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
         Add a[i] to S;
         i :=m;} return S;
}
```

Example:

Already sorted

according to finish time.

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

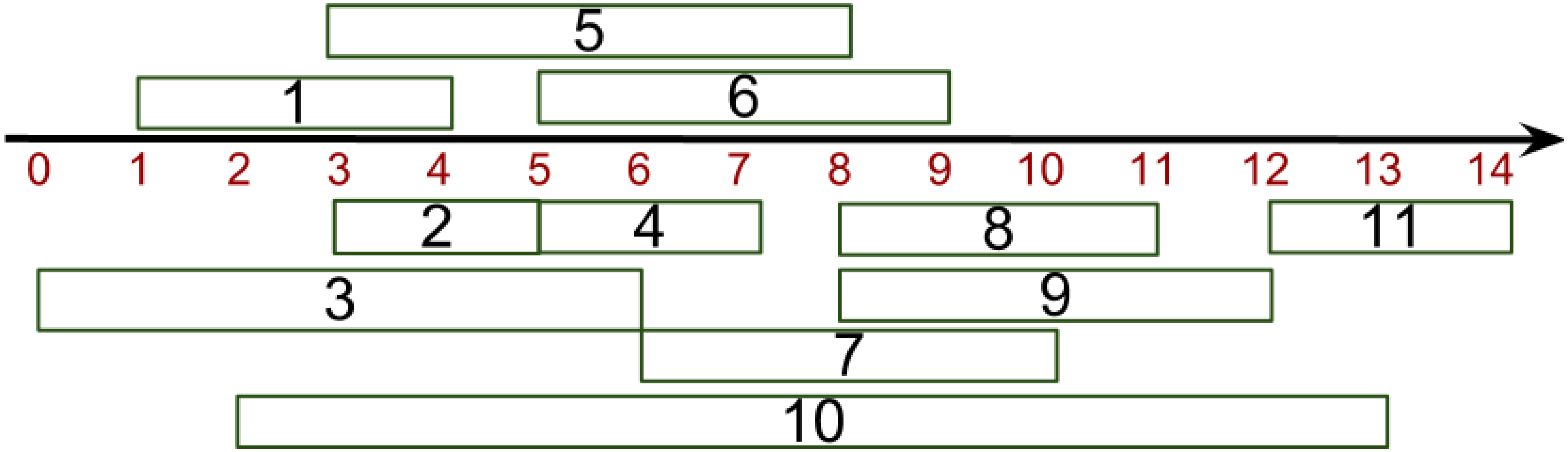


```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



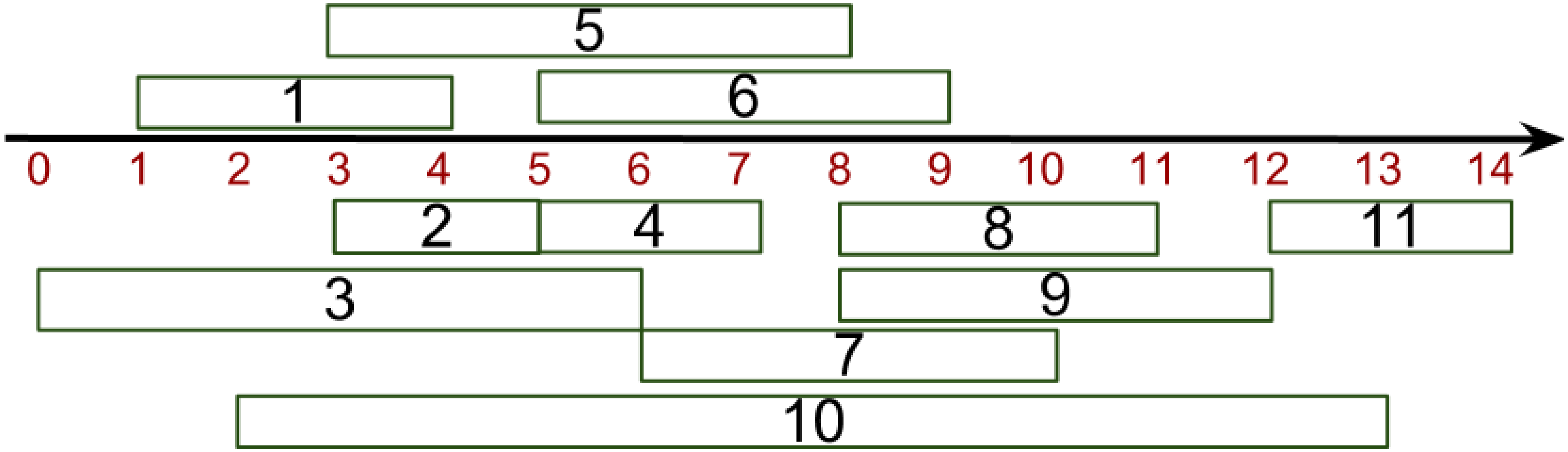
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
         Add a[i] to S;
         i :=m;} return S;
}
```

Example:

$S:=\{a_1\}$

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

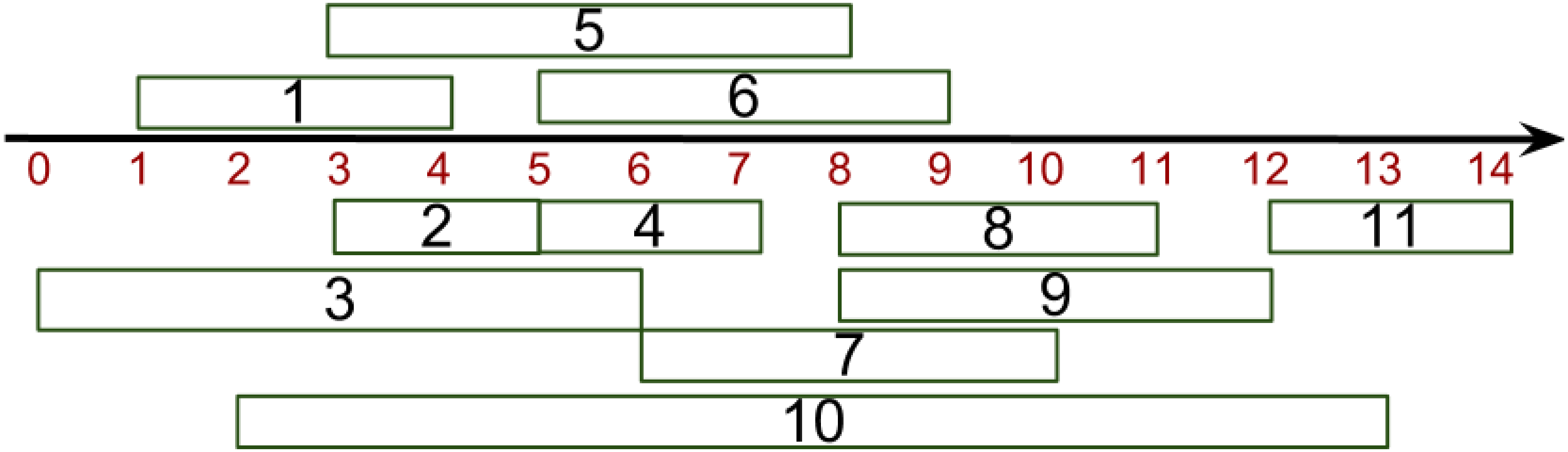
Example:

$S:=\{a_1\}$

i

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

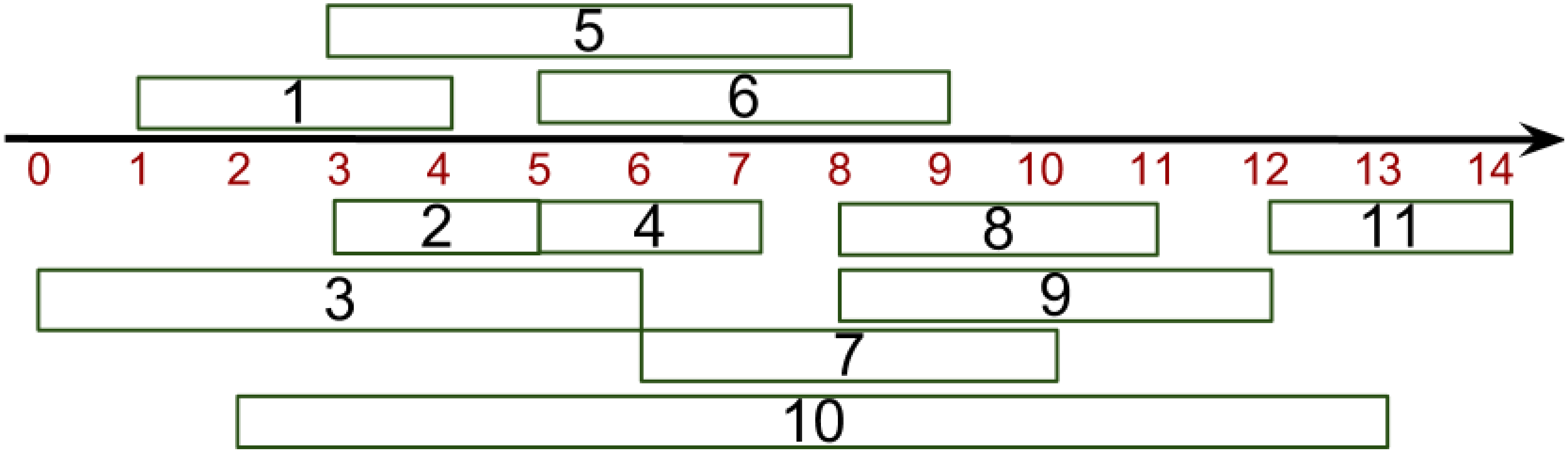
Example:

S:=$\{a_1\}$

i m

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

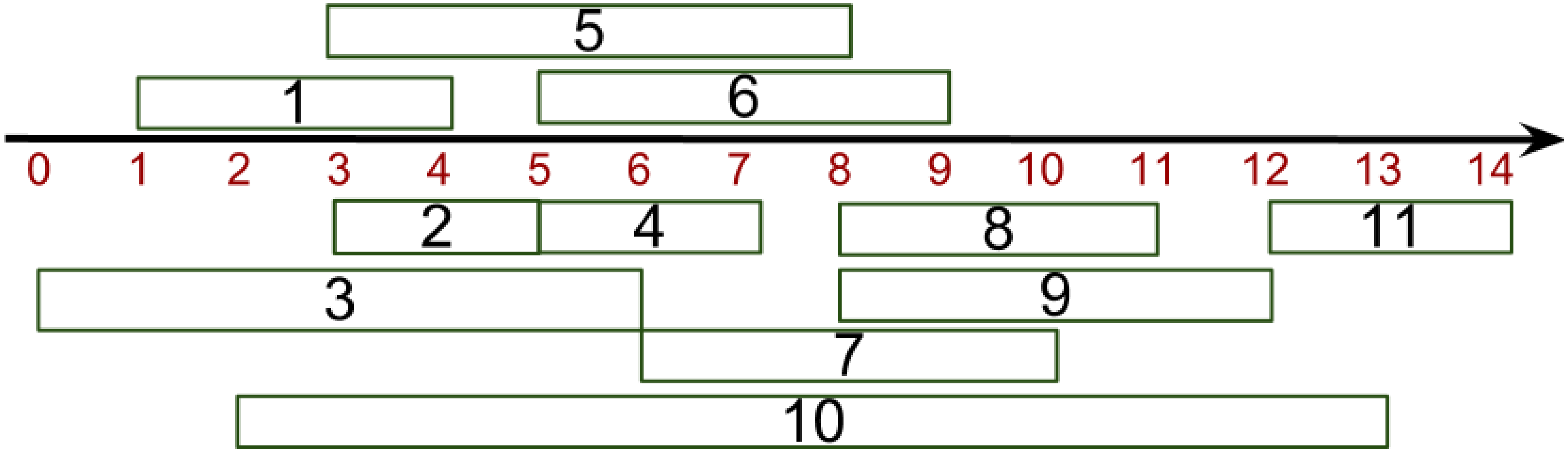
$S:=\{a_1\}$

$s[2] \geq f[1]$?

i m

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

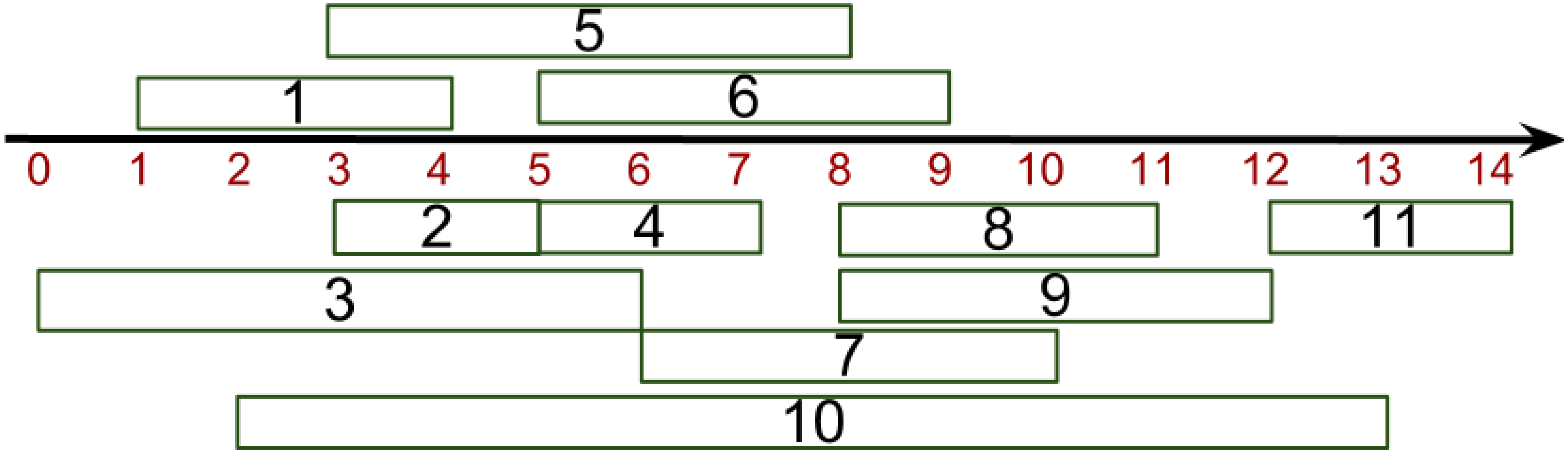
Example:

$S:=\{a_1\}$

s[2] < f[1]

i   m   n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
      Add a[i] to S;
      i :=m;} return S;
}
```

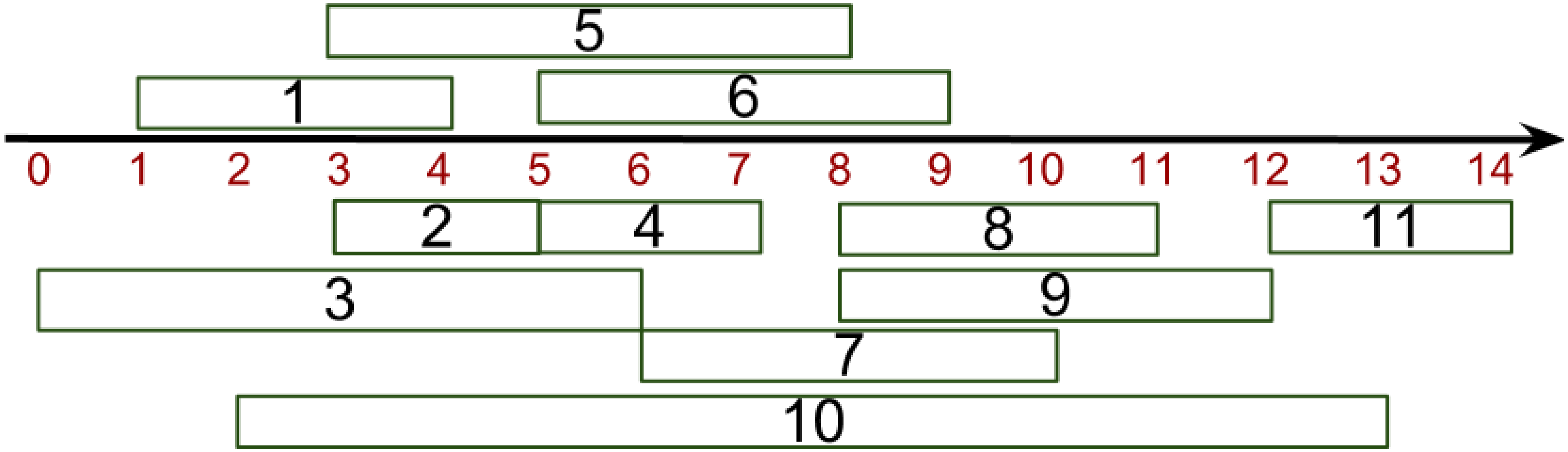
Example:

$S:=\{a_1\}$

$s[3] \geq f[1]$?

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

Example:

$S:=\{a_1\}$

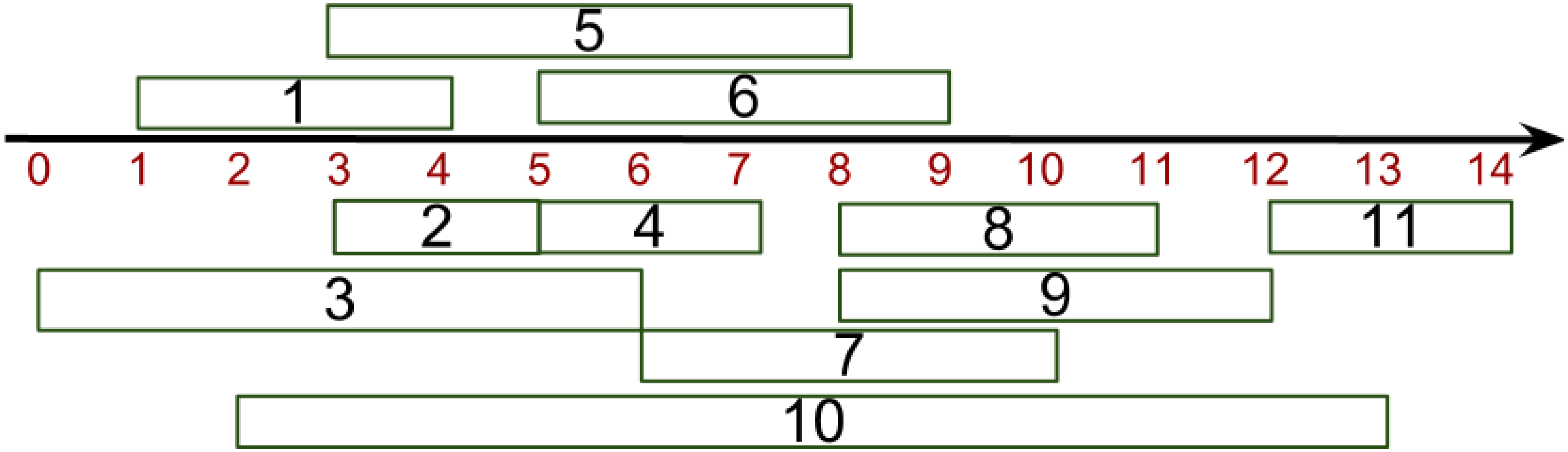
s[3] < f[1]

i

m

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

$S:=\{a_1\}$

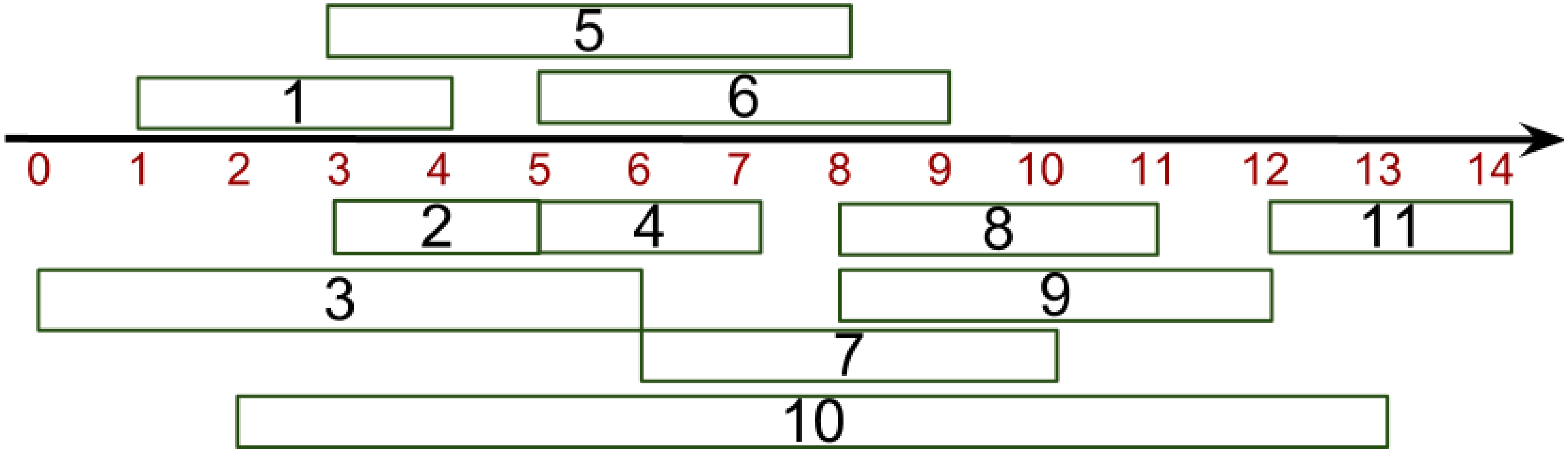
$s[4] \geq f[1]$?

i

m

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

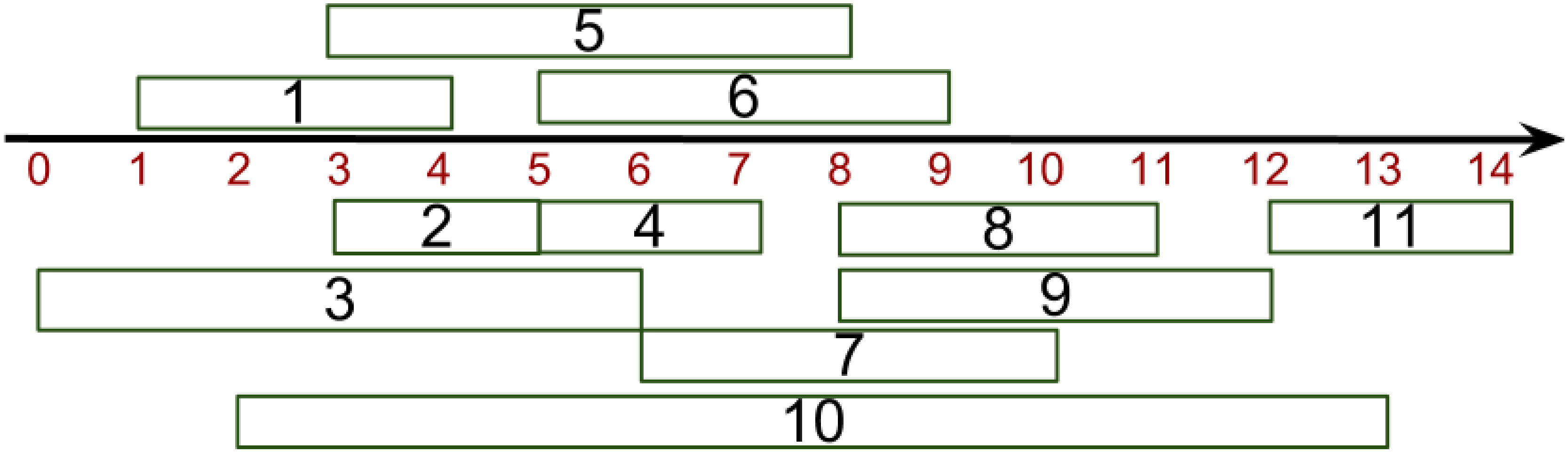
Example:

$S:=\{a_1, a_4\}$

s[4] > f[1]

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

Example:

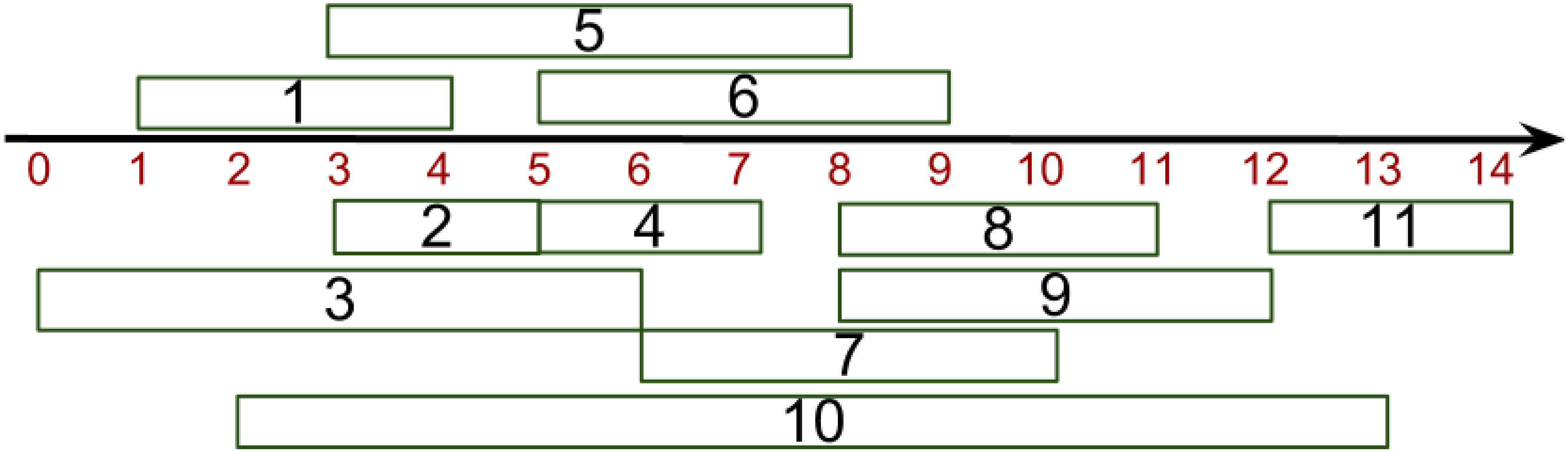
$S:=\{a_1, a_4\}$

s[4] > f[1]

i,m

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
         Add a[i] to S;
         i :=m;} return S;
}
```

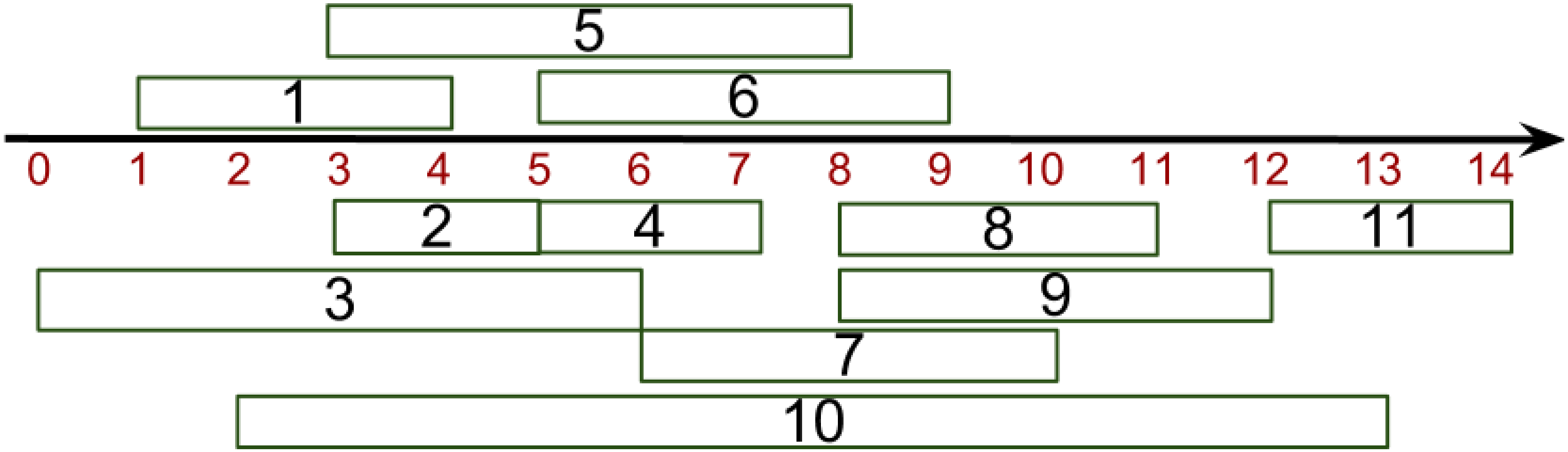
Example:

$S:=\{a_1, a_4\}$

i m

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

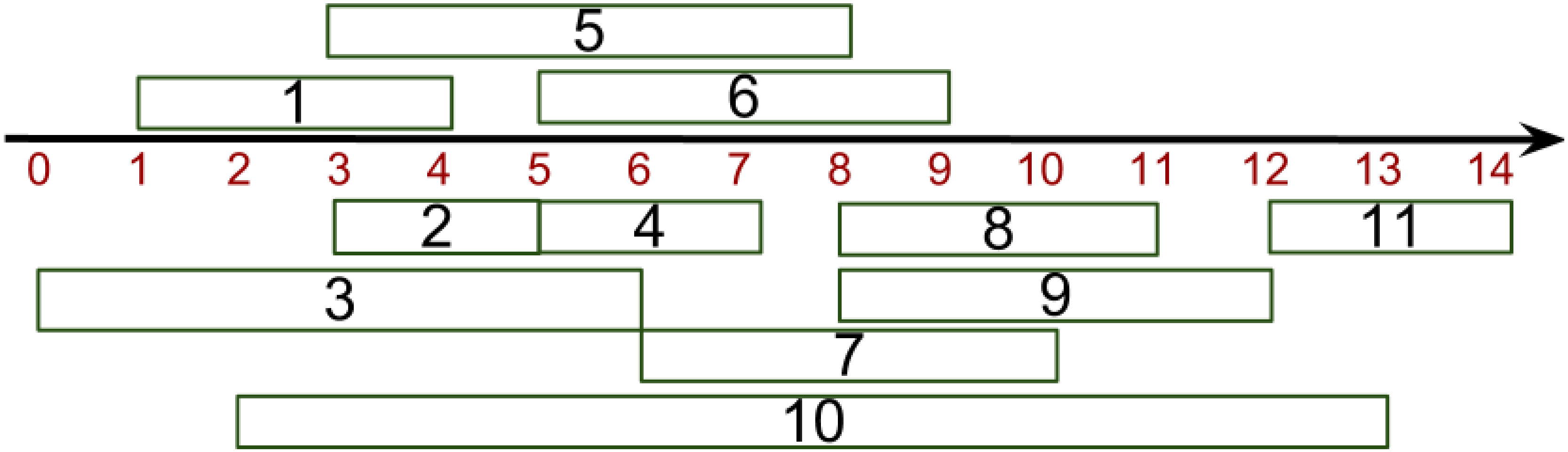
$S:=\{a_1 ,a_4\}$

$s[5] \geq f[4]$?

i m

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

Example:

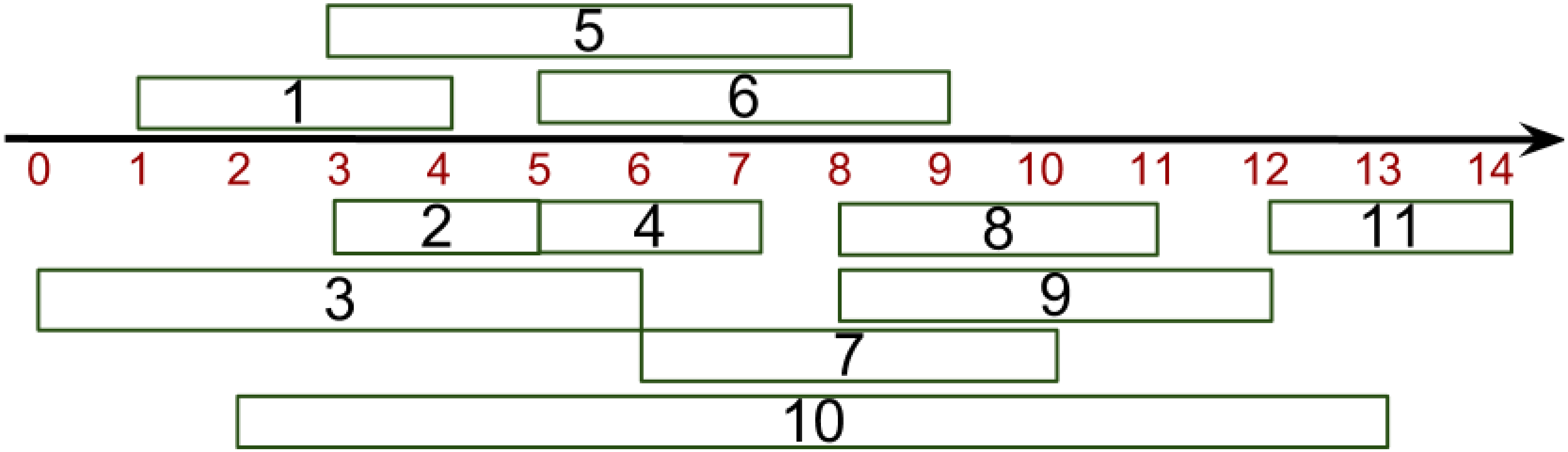
$S:=\{a_1 ,a_4\}$

s[5] < f[4]

i ↓ (column 4) m ↓ (column 6)

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

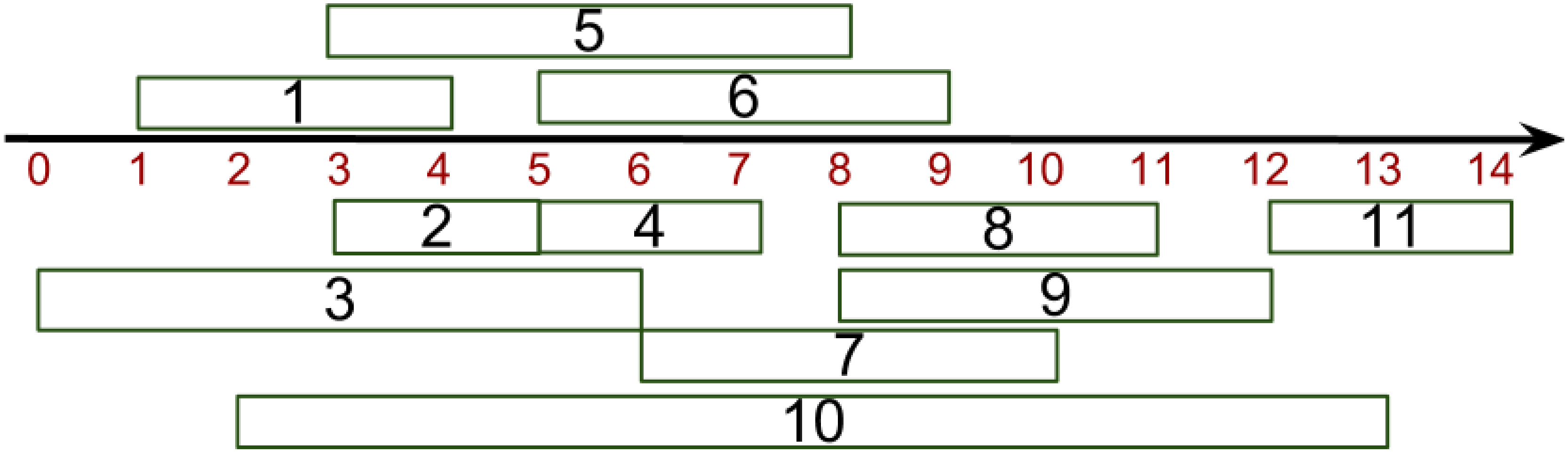
Example:

$S:=\{a_1, a_4\}$

$s[6] \geq f[4]$?

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

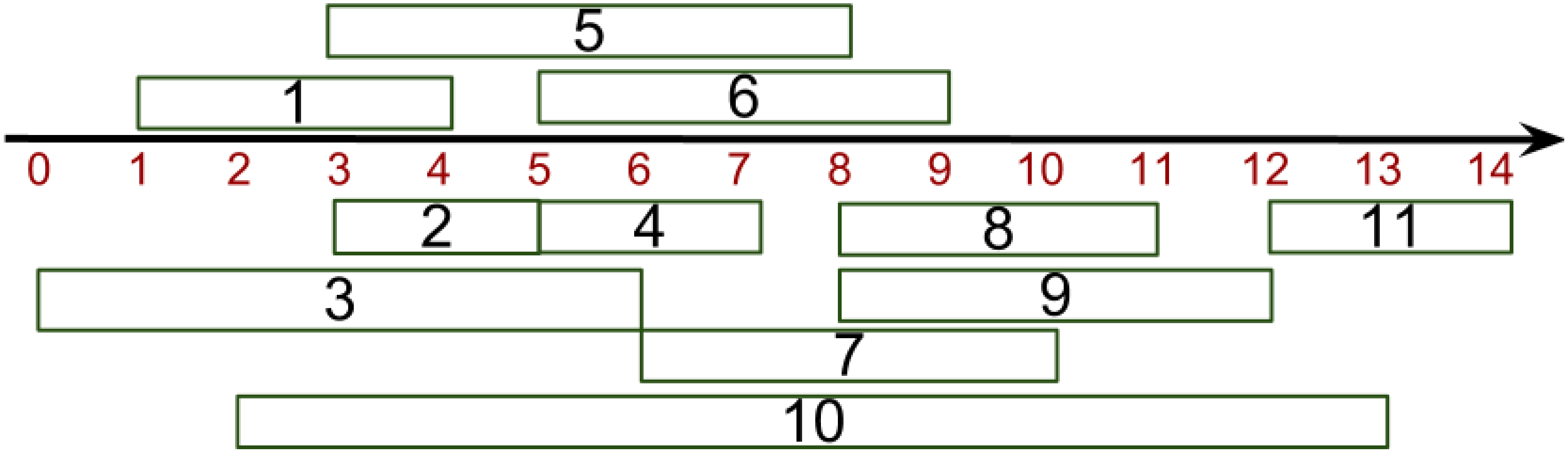
Example:

$S:=\{a_1, a_4\}$

s[6] < f[4]

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
      Add a[i] to S;
      i :=m;} return S;
}
```

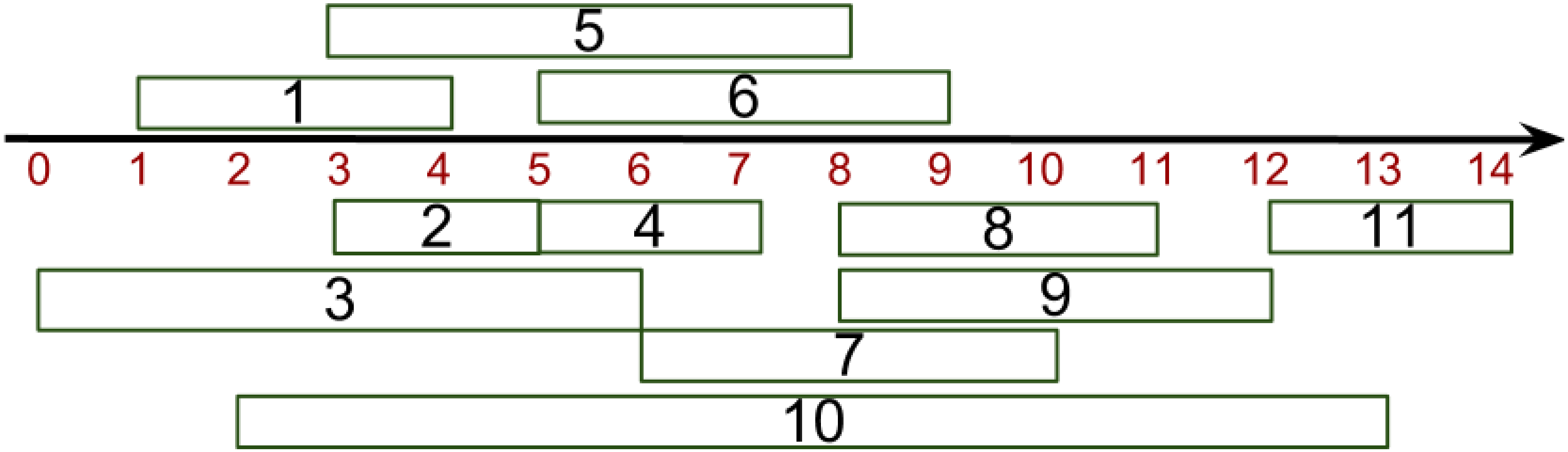
Example:

$S:=\{a_1 ,a_4\}$

$s[7] \geq f[4]$?

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
         Add a[i] to S;
         i :=m;} return S;
}
```

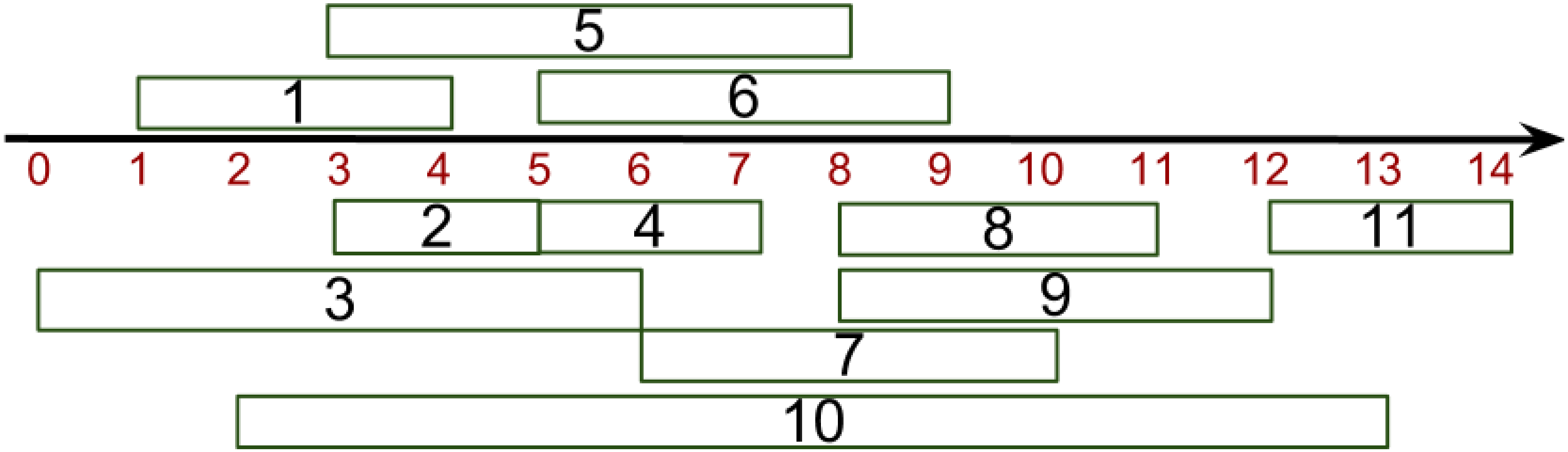
Example:

$S:=\{a_1, a_4\}$

$s[7] < f[4]$

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

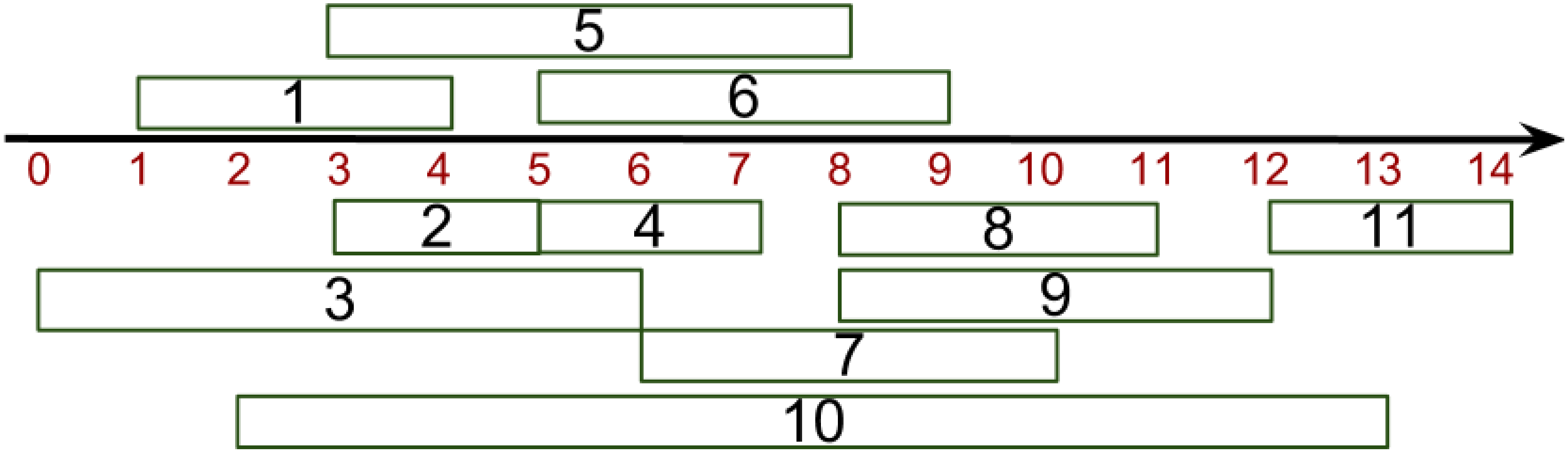
Example:

$S:=\{a_1 ,a_4\}$

s[8] ≥ f[4] ?

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

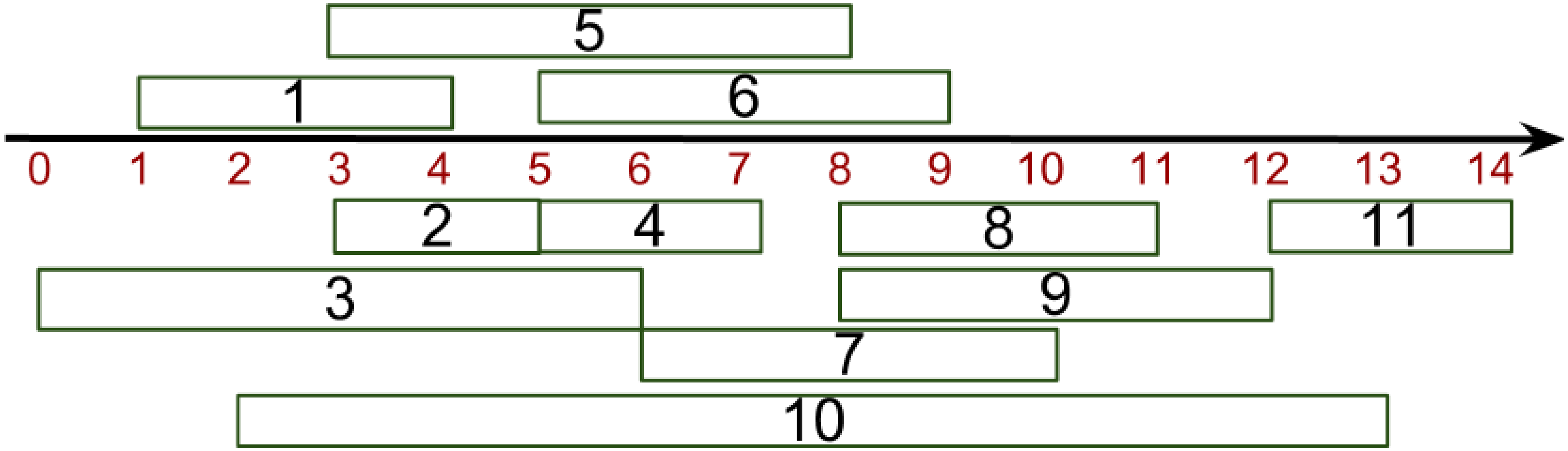
Example:

$S:=\{a_1, a_4, a_8\}$

s[8] > f[4]

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
         Add a[i] to S;
         i :=m;} return S;
}
```

Example:

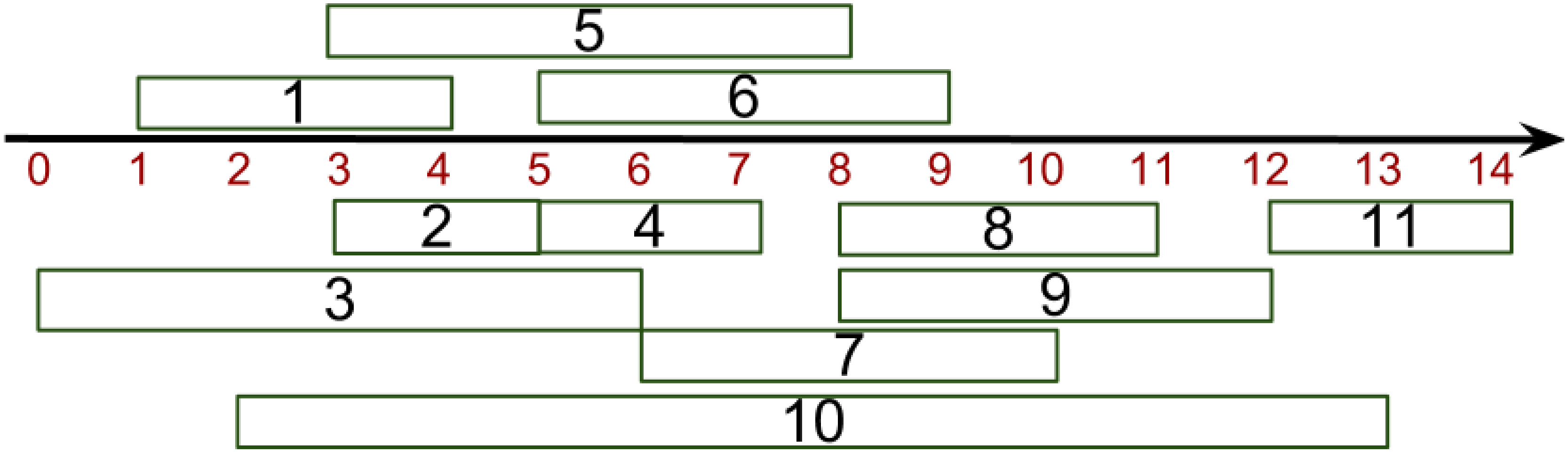
$S:=\{a_1 ,a_4 ,a_8\}$

s[8] > f[4]

i,m

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



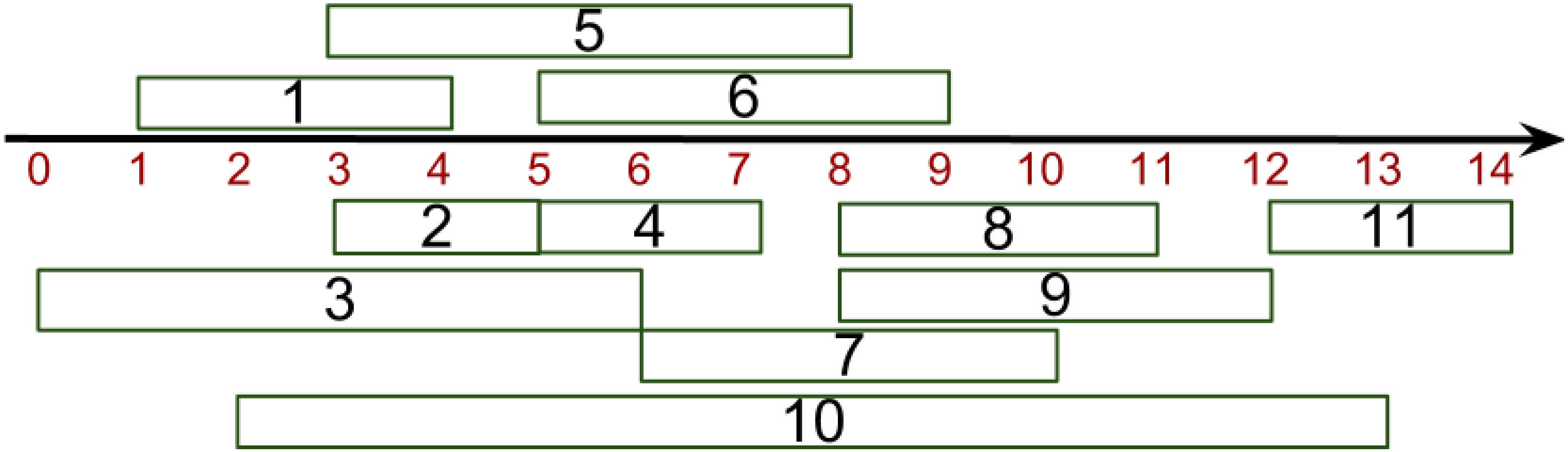
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

$S:=\{a_1, a_4, a_8\}$

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

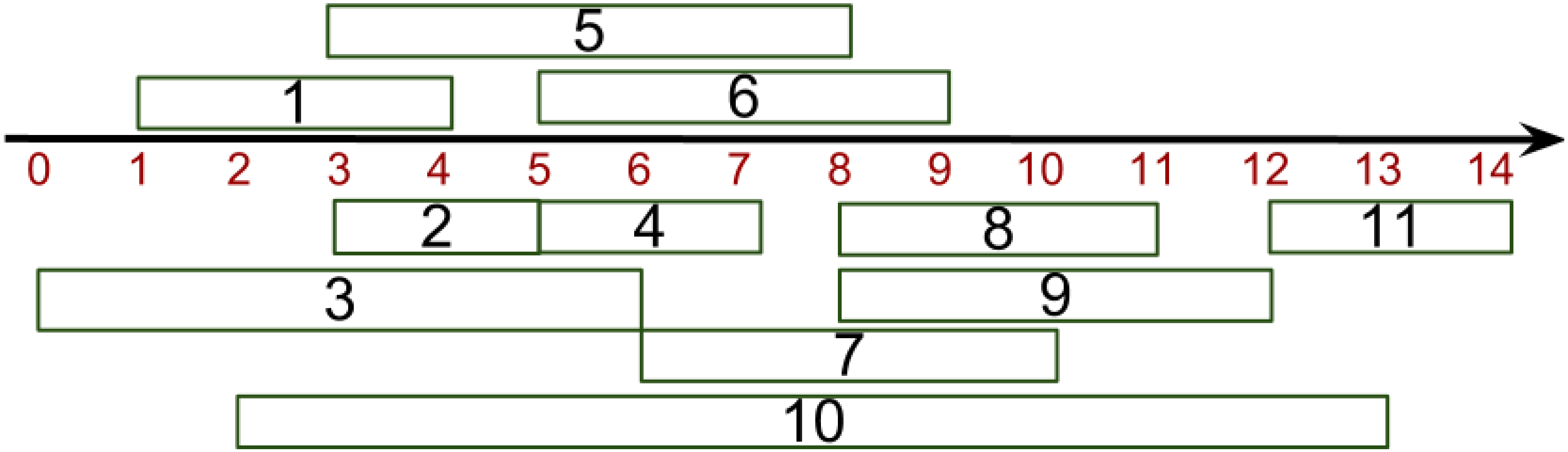
Example:

$S:=\{a_1 ,a_4 ,a_8\}$

$s[9] \geq f[8]$?

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

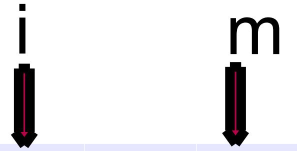


```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

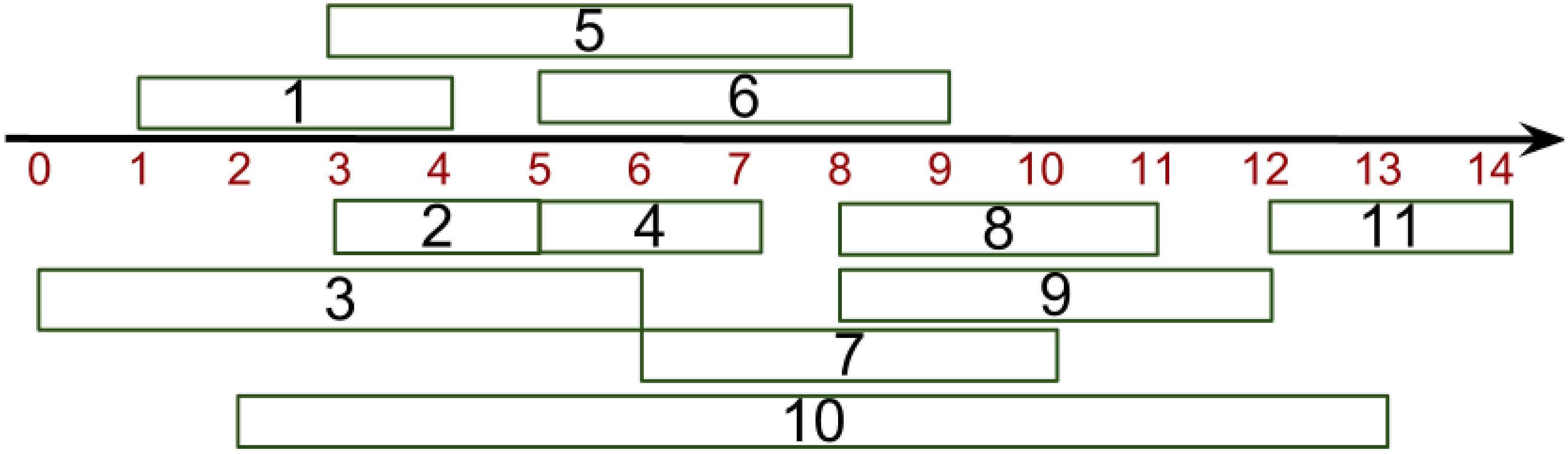
Example:

$S:=\{a_1 ,a_4 ,a_8\}$

$s[9] < f[8]$



a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



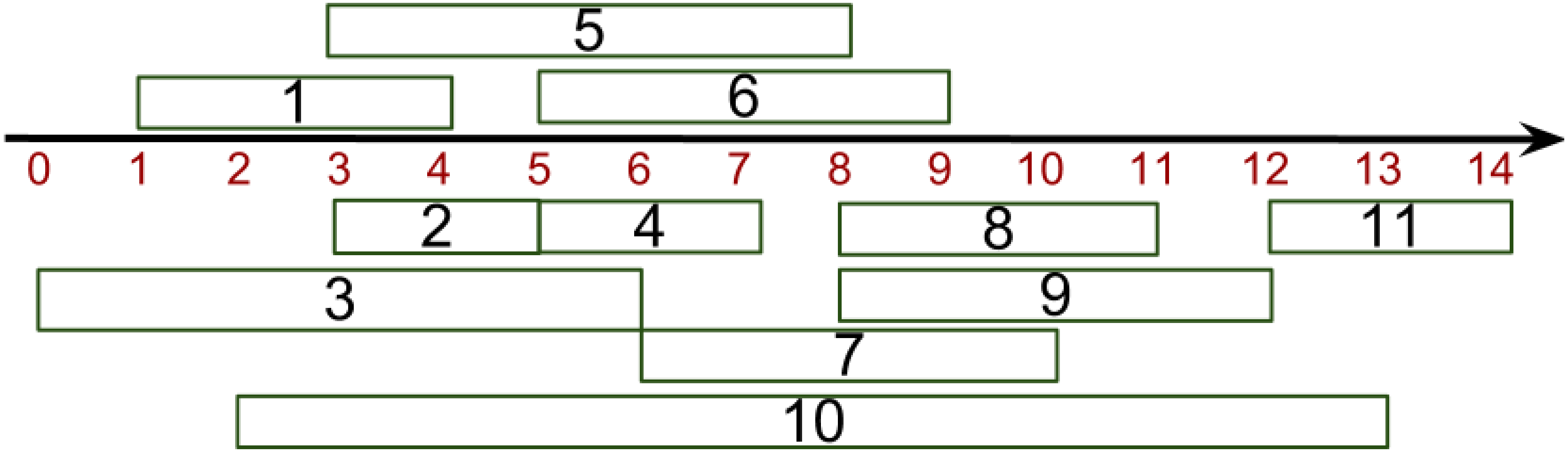
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

S:={a_1 ,a_4 ,a_8}

s[10] ≥ f[8] ?

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



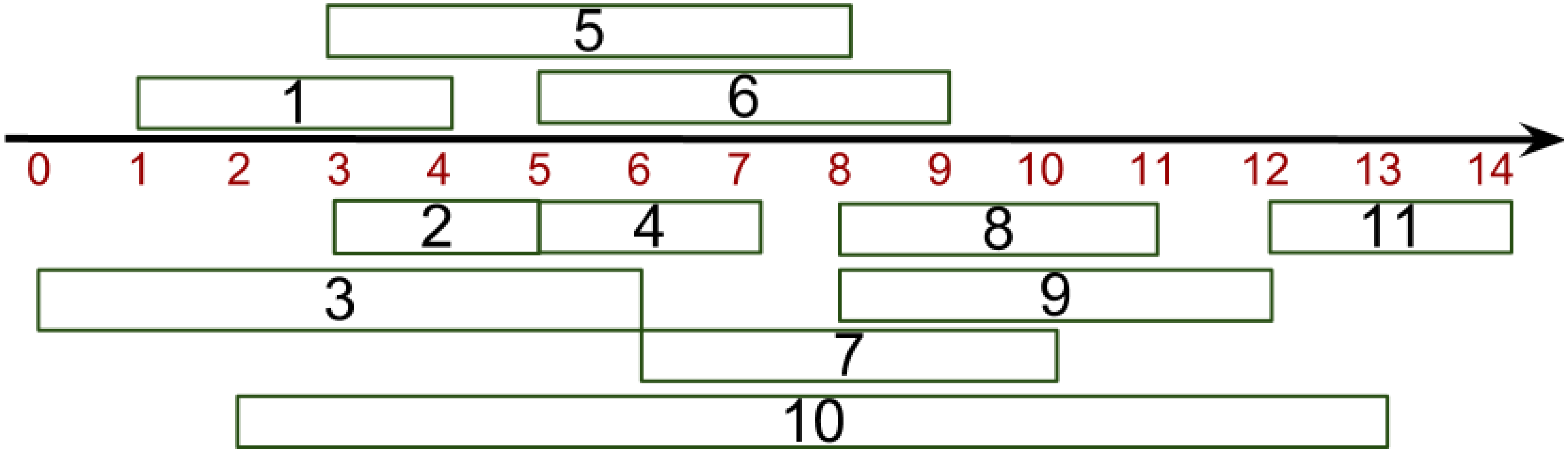
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

S:={a_1 ,a_4 ,a_8}

s[10] < f[8]

								i			m
a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

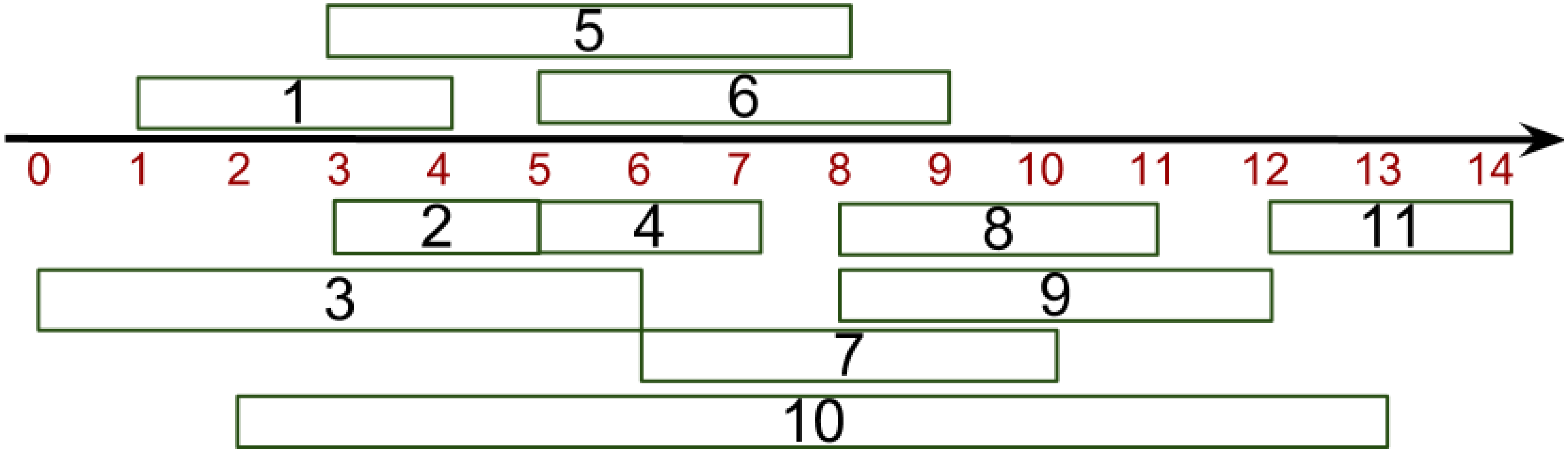
Example:

S:={a_1 ,a_4 ,a_8}

S[11] ≥ f[8] ?

i m

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



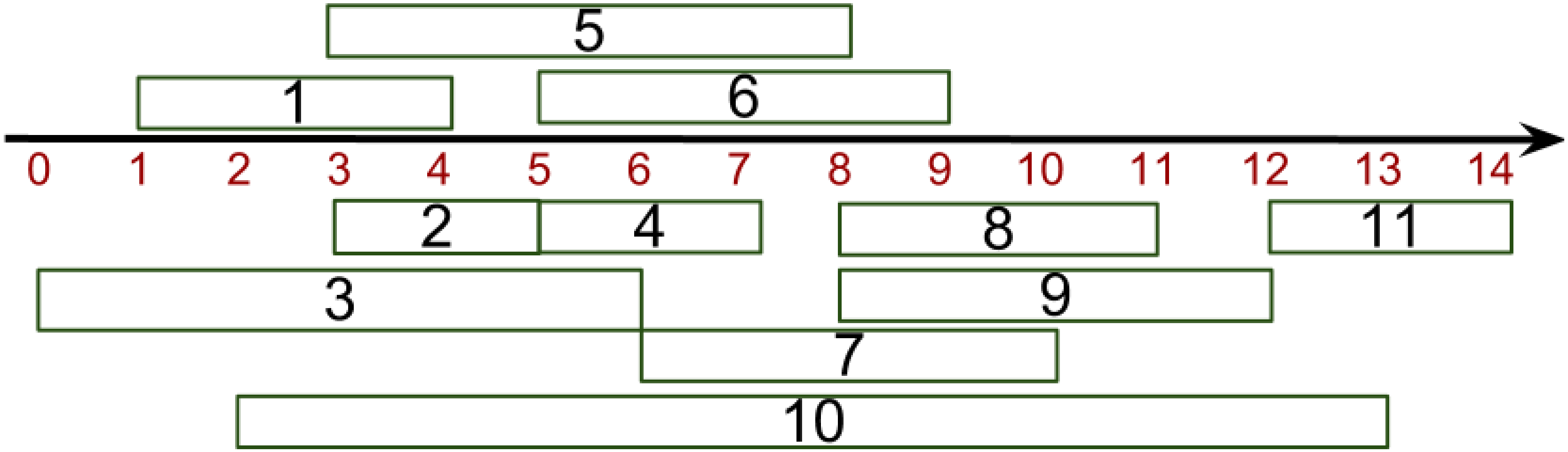
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

$S:=\{a_1, a_4, a_8, a_{11}\}$

$s[11] \geq f[8]$

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

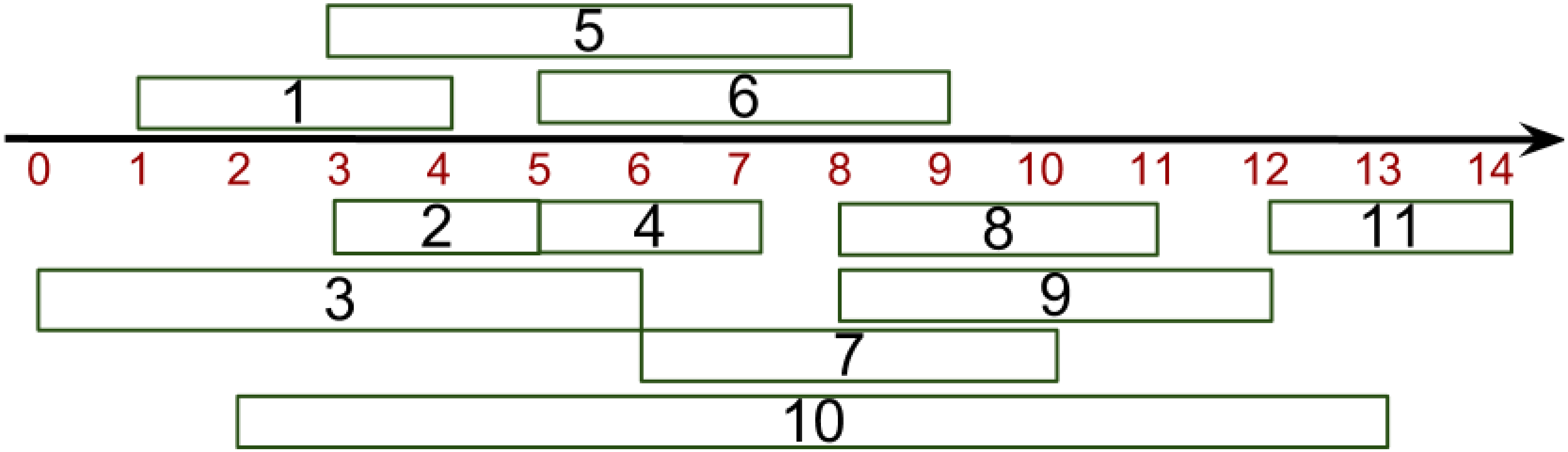
Example:

$S:=\{a_1, a_4, a_8, a_{11}\}$

$s[11] \geq f[8]$

i,m

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

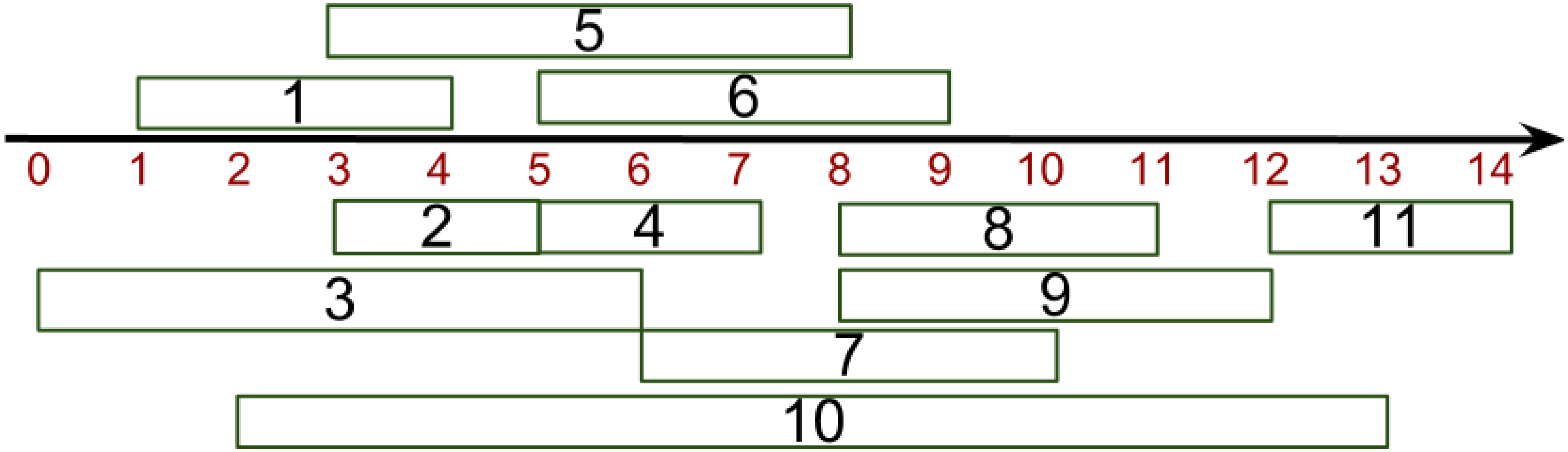
$S:=\{a_1 ,a_4 ,a_8 ,a_{11}\}$

m = 12,

n = 11

i

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

Example:

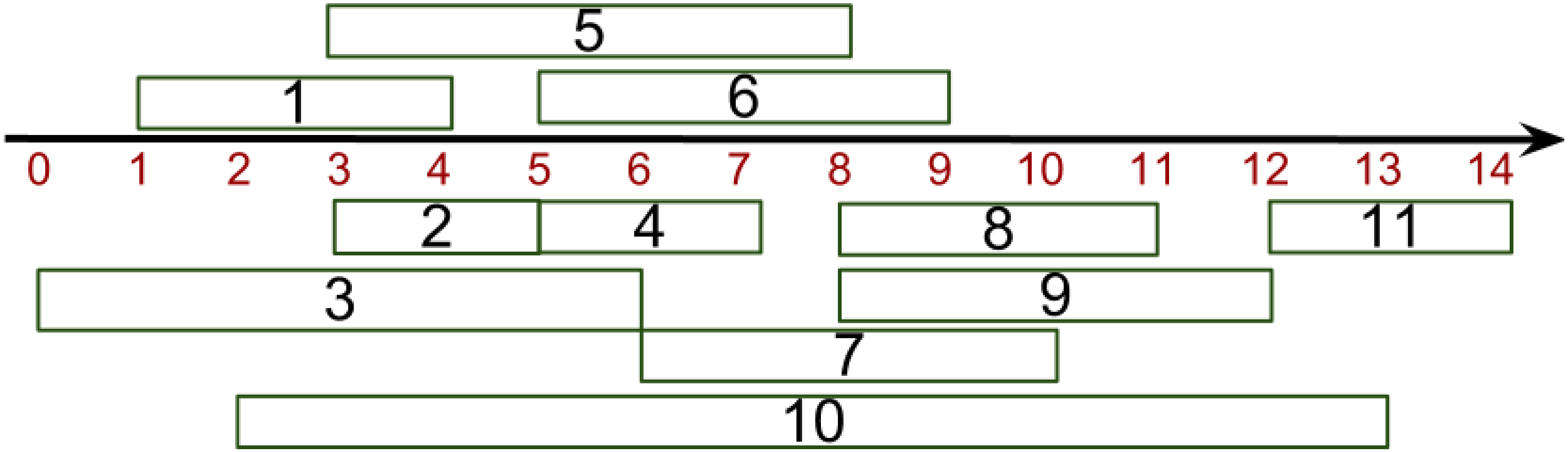
$S:=\{a_1 ,a_4 ,a_8 ,a_{11}\}$

m = 12,

n = 11

i

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

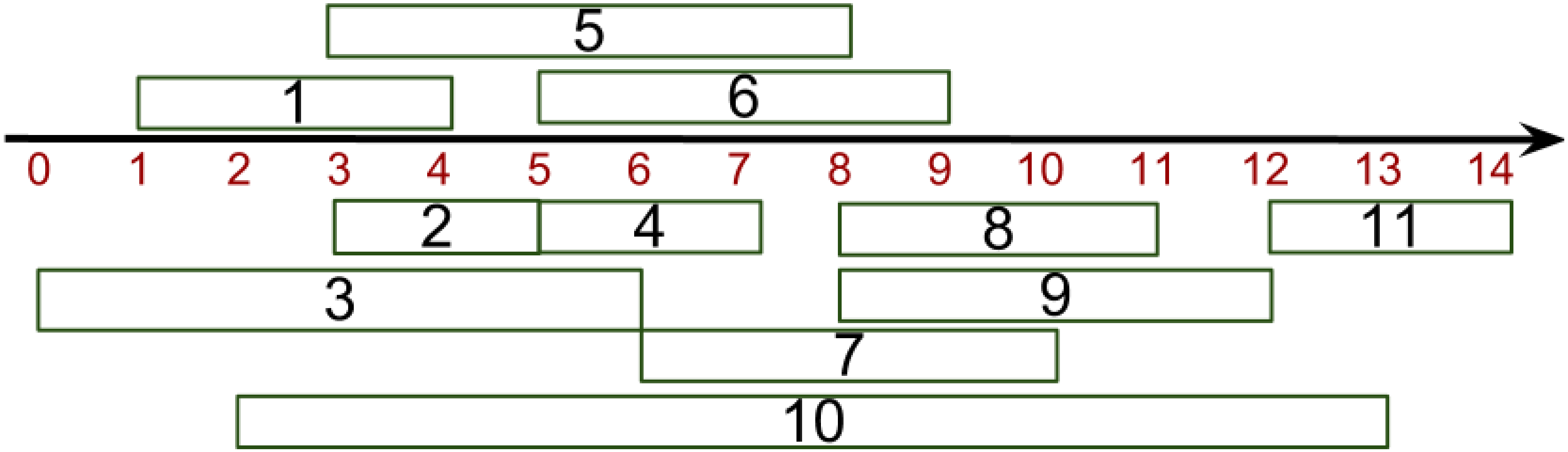


```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

$S:=\{a_1 ,a_4 ,a_8 ,a_{11}\}$

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



- Fractional knapsack
- Values v_i weights w_i

- Sort according to v_i / w_i

- Pick as much as possible of item with max v_i / w_i , and so on