Dynamic programming

“Life can only be understood backwards;
but it must be lived forwards."

Soren Kierkegaard

Dynamic programming

An interesting question is, "Where did the name, dynamic programming, come from?" The 1950's were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defence, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place, I was interested in planning, in decision-making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying- I thought, let's kill two birds with one stone. Let's take a word which has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in the pejorative sense. Try thinking of some combination which will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

Richard Bellman, Eye of the Hurricane an autobiography, p. 159

The coin change problem

- You are a cashier and have k infinite piles of coins with values d_1 , ... , d_k
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Example:
  k = 3, d = (5,4,1), t = 8

  One solution has cost 4: t = 5+1+1+1
  A better solution has cost 2: t = 4+4, which is optimal

  Cost[t] = 2.

The coin change problem

- You are a cashier and have k infinite piles of coins with values d_1 , ... , d_k
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Try to obtain a recursion:

The coin change problem

- You are a cashier and have k infinite piles of coins with values d_1 , ... , d_k
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Try to obtain a recursion: To give change for t you can:

The coin change problem

- You are a cashier and have k infinite piles of coins with values d_1 , ... , d_k
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Try to obtain a recursion: To give change for t you can:
  use coin d_1, then need change for $t - d_1 \Rightarrow$ $Cost[t] \leq 1+Cost[t - d_1]$

The coin change problem

- You are a cashier and have k infinite piles of coins with values d_1 , ... , d_k
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Try to obtain a recursion: To give change for t you can:
  use coin d_1, then need change for $t - d_1 \Rightarrow$ $\text{Cost}[t] \leq 1+\text{Cost}[t - d_1]$
  or use coin d_2, then need change for $t - d_2 \Rightarrow$ $\text{Cost}[t] \leq 1+\text{Cost}[t - d_2]$

The coin change problem

- You are a cashier and have k infinite piles of coins with values $d_1 , \ldots , d_k$
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Try to obtain a recursion: To give change for t you can:
  use coin d_1, then need change for $t – d_1 \Rightarrow$ $Cost[t] \leq 1+Cost[t – d_1]$
  or use coin d_2, then need change for $t – d_2 \Rightarrow$ $Cost[t] \leq 1+Cost[t – d_2]$
  or ....

  Which one to pick?

The coin change problem

- You are a cashier and have k infinite piles of coins with values $d_1, \ldots, d_k$
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Try to obtain a recursion: To give change for t you can:
  use coin d_1, then need change for $t - d_1 \Rightarrow$ $\text{Cost}[t] \leq 1+\text{Cost}[t - d_1]$
  or use coin d_2, then need change for $t - d_2 \Rightarrow$ $\text{Cost}[t] \leq 1+\text{Cost}[t - d_2]$
  or ….

  Which one to pick? The one that gives the minimum:

$$\boxed{\text{Cost}[t] = 1 + \min_{i \leq k} \text{Cost}[t - d_i]}$$

The coin change problem

- You are a cashier and have k infinite piles of coins with values $d_1 , \dots , d_k$
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Recursion $\boxed{\text{Cost}[t] = 1 + \min_{i \le k} \text{Cost}[t - d_i]}$

The coin change problem

- You are a cashier and have k infinite piles of coins with values d_1 , ... , d_k
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Recursion $\boxed{\text{Cost}[t] = 1 + \min_{i \leq k} \text{Cost}[t - d_i]}$

- A false start: a naive recursive algorithm

```
Alg(t) {
        return min_{i ≤ k} Alg(t - d_i )
    }
```

The coin change problem

- You are a cashier and have k infinite piles of coins with values $d_1 , \ldots , d_k$
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Recursion $\boxed{\text{Cost}[t] = 1 + \min_{i \le k} \text{Cost}[t - d_i]}$

- A false start: a naive recursive algorithm

```
Alg(t) {
        return min i ≤ k Alg(t - d_i )
      }
```

- Running time of Alg, even for $k = 2$, $d_1 = 1$, $d_2 = 2$

- $T(t) \ge T(t-1)+T(t-2) \ge T(t-2)+T(t-3)+T(t-2) \ge 2T(t-2)$

The coin change problem

- You are a cashier and have k infinite piles of coins with values d_1 , ... , d_k
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Recursion $\boxed{\text{Cost}[t] = 1 + \min_{i \leq k} \text{Cost}[t - d_i]}$

- A false start: a naive recursive algorithm

```
Alg(t) {
        return min_{i ≤ k} Alg(t - d_i)
    }
```

- Running time of Alg, even for $k = 2$, $d_1 = 1$, $d_2 = 2$

- $T(t) \geq T(t-1)+T(t-2) \geq T(t-2)+T(t-3)+T(t-2) \geq 2T(t-2) \Rightarrow T(t) \geq 2^{t/2}$

The coin change problem

- You are a cashier an have finit es oin th es d ... , d_k
  You have to gi chang
  You want to use fi

- Definition: C

- Recur

Stop solving over and over again the same problems!!!

For example, below you are recursing multiple times on problem Cost[t-2]. You should only compute this once!

- A false start: c

- Running time of Alg, even fo k = 2, = 1, = 2

- $T(t) \geq T(t-1)+T(t-2) \geq T(t-2)+T(t-3)+T(t-2) \geq 2T(t-2) \Rightarrow T(t) \geq 2^{t/2}$

The coin change problem

- You are a cashier and have k infinite piles of coins with values $d_1, \ldots, d_k$
  You have to give change for t
  You want to use the minimum number of coins

- Definition: Cost[t] := minimum number of coins to obtain t

- Recursion $\boxed{\text{Cost}[t] = 1 + \min_{i \leq k} \text{Cost}[t - d_i]}$

- Alg(t): { Auxiliary array C[0..t]
  C[0] = 0
  For (s = 1..t) {
  m = minimum of C[s - d_i] over i = 1..k such that s – $d_i \geq 0$
  C[S] = 1 + m
  }

- Running time: O(t k)

The coin change problem

- Example:
  k = 3, d = (5,4,1), t = 8

	0	1	2	3	4	5	6	7	8
C	0								

The coin change problem

- Example:
  k = 3, d = (5,4,1), t = 8

	0	1	2	3	4	5	6	7	8
C	0	1							

Cost[1] = 1 + Cost[0]

The coin change problem

- Example:
  k = 3, d = (5,4,1), t = 8

	0	1	2	3	4	5	6	7	8
C	0	1	2						

Cost[2] = 1 + Cost[1]

The coin change problem

- Example:
  k = 3, d = (5,4,1), t = 8

	0	1	2	3	4	5	6	7	8
C	0	1	2	3					

Cost[3] = 1 + Cost[2]

The coin change problem

- Example:
  k = 3, d = (5,4,1), t = 8

	0	1	2	3	4	5	6	7	8
C	0	1	2	3	1				

Cost[4] = 1 + Minimum(Cost[3], Cost[0]) = 1 + Minimum(3,0) = 1

The coin change problem

- **Example**:
  k = 3, d = (5,4,1), t = 8

	0	1	2	3	4	5	6	7	8
C	0	1	2	3	1	1			

Cost[5] = 1 + Minimum(Cost[4],Cost[1],Cost[0]) = 1 + Minimum(1,1,0) = 1

The coin change problem

- Example:

  k = 3, d = (5,4,1), t = 8

	0	1	2	3	4	5	6	7	8
C	0	1	2	3	1	1	2		

Cost[6] = 1 + Minimum(Cost[5],Cost[2],Cost[1]) = 1 + Minimum(1,2,1) = 2

The coin change problem

- Example:
  k = 3, d = (5,4,1), t = 8

	0	1	2	3	4	5	6	7	8
C	0	1	2	3	1	1	2	3	

Cost[7] = 1 + Minimum(Cost[6],Cost[3],Cost[2]) = 1 + Minimum(2,3,2) = 3

The coin change problem

- Example:
  $k = 3$, $d = (5,4,1)$, $t = 8$

	0	1	2	3	4	5	6	7	8
C	0	1	2	3	1	1	2	3	2

Cost[8] = 1 + Minimum(Cost[7],Cost[4],Cost[3]) = 1 + Minimum(3,1,3) = 2

The coin change problem

- So far we computed how many coins
- Now want to know which values, as in 8 = 4+4
- Alg2(t): { Auxiliary arrays C[0..t], A[0..t]
  C[0] = 0; A[0] = 0
  For (s = 1..t) {
  m = minimum of C[s - d_i] over i = 1..k such that s – $d_i \geq 0$
  i = arg-minimum
  C[s] = 1 + m
  A[s] = d_i
  }
- Idea: values are: A[t], A[t-A[t]], …. until you get zero

The coin change problem

- Printing the coins used

- Print-Coins(t) {

```
    for(i = t; i > 0; i = i – A[i] )
        Print(A[i])
}
```

- Time O(t)

The coin change problem

- Example:
  k = 3, d = (5,4,1), t = 8

	0	1	2	3	4	5	6	7	8
C	0	1	2	3	1	1	2	3	2
A	0	1	1	1	4	5	5	5	4

Print-coins(8) = 4, 4

The coin change problem

- Example:
  k = 3, d = (5,4,1), t = 8

	0	1	2	3	4	5	6	7	8
C	0	1	2	3	1	1	2	3	2
A	0	1	1	1	4	5	5	5	4

Print-coins(7) = 5, 1, 1

Steps for dynamic programming

- Identify subproblems (In coin-change example Cost[1..t])
- Obtain recursion (Cost[t] = 1 + min $_{i \leq k}$ Cost[t - d_i])
  (aka structure of solutions, optimal substructure property)
- Algorithm solves all the subproblems, once
- Running time = Number of subproblems (here t)
  x Time to compute recursion (here O(k))

- Saw dynamic programming as iterative, “bottom-up”: solve all the problems from the smallest to the biggest.

- Can also be implemented in a “top-down” recursive fashion: 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.

Longest common subsequence

- Given two strings X and Y over some alphabet, want to find a longest subsequence Z.

  The symbols in Z appear in X, Y in the same order,

  but not necessarily consecutively

- Example: Alphabet = {A, C, G, T}

  X = A A G G A C A C T C T A G C G A T

  Y = T G G C A T T T A C G C G C A A

Longest common subsequence

- Given two strings X and Y over some alphabet, want to find a longest subsequence Z.

  The symbols in Z appear in X, Y in the same order,

  but not necessarily consecutively

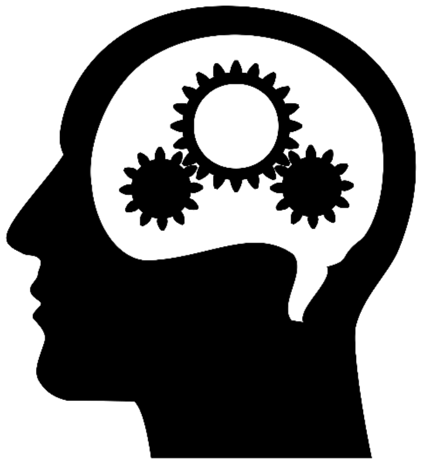
- Example: Alphabet = {A, C, G, T}

  X = A A G G A C A C T C T A G C G A T

  Y = T G G C A T T T A C G C G C A A

  Z = G A T T A C A

Longest common subsequence



- Arriving at subproblems and recursion

  X = A A G G A C A C T C T A G C G A T
  Y = T G G C A T T T A C G C G C A A

The strings X and Y end with different symbols.
So either last T in X is not part of the solution,
or last A in Y is not part of the solution.

In the first case I can remove last T from X
Now both strings end with A, which can be matched.

In the latter case I can remove the last A from Y.

Longest common subsequence

- On input $X[1..m]$, $Y[1..n]$,
  consider the prefixes $X[1..i]$, $Y[1..j]$ for any $i \le m$, $j \le n$.

- Subproblems:
  $L(i,j)$ = length longest subsequence of $X[1..i]$ and $Y[1..j]$

- Recursion:

if $i = 0$ or $j = 0$	$L(i,j) = 0$
if $X[i] = Y[j]$	$L(i,j) = ?$
if $X[i] \neq Y[j]$	$L(i,j) = ?$

Longest common subsequence

- On input X[1..m], Y[1..n],
  consider the prefixes X[1..i], Y[1..j] for any $i \leq m$, $j \leq n$.

- Subproblems:
  L(i,j) = length longest subsequence of X[1..i] and Y[1..j]

- Recursion:

if i = 0 or j = 0	L(i,j) = 0
if X[i] = Y[j]	L(i,j) = L(i-1,j-1) + 1
if X[i] ≠ Y[j]	L(i,j) = ?

Longest common subsequence

- On input X[1..m], Y[1..n],
  consider the prefixes X[1..i], Y[1..j] for any $i \leq m$, $j \leq n$.

- Subproblems:
  L(i,j) = length longest subsequence of X[1..i] and Y[1..j]

- Recursion:

if $i = 0$ or $j = 0$	$L(i,j) = 0$
if $X[i] = Y[j]$	$L(i,j) = L(i-1,j-1) + 1$
if $X[i] \neq Y[j]$	$L(i,j) = \max \{L(i-1,j), L(i,j-1)\}$

- LCSLength(X[1..m], Y[1..n])

```
L = zero array(0..m, 0..n)

for i := 1..m
  for j := 1..n
    if X[i] = Y[j]
      L[i,j] := L[i-1,j-1] + 1
    else
      L[i,j] := max(L[i,j-1], L[i-1,j])

return L[m,n]
```

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	C	U
0	Ø	0	0	0	0	0	0	0	0
1	C	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	T	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

- Running time = O(mn)

Longest common subsequence

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

  ↖ if the last symbols match
  ← if we are dropping last symbol of X
  ↑ if we are dropping last symbol of Y

  Then we can reconstruct
  the sequence backwards.

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	C	U
0	Ø	0	0	0	0	0	0	0	0
1	C	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	T	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

M
J
A
U

Dynamic programming in economics

- Let us plan Bob's next L years.
- He has \$w, and every year makes \$w
- At the beginning of each year,
  he must decide how much to consume, rest is saved
    Savings earn interest (1+ρ)   (round to integer)
    Consuming C yields utility log(C)
    (\$10K vs. \$20K is different from \$1M vs. \$1M+\$10K)
- He wants to maximize sum of utility throughout his lifetime

Life can only be understood backwards;
but it must be lived forwards.

Soren Kierkegaard

- Subproblems and recursion

- U[k,i] := utility for years i, i+1, …, L if at beginning of year i have $k. Note k integer $\leq M := wL(1 + \rho)^L$

- U[k,L] := ?

How much should Bob consume in his last year of life?

- Subproblems and recursion

- U[k,i] := utility for years i, i+1, …, L if at beginning of year i have \$k. Note k integer $\leq M := wL\,(1 + \rho)^L$

- U[k,L] := log(k)
  Consumption = k, because at last year L he spends all

- U[k,i] := What recursion for i < L ?

- Subproblems and recursion

- U[k,i] := utility for years i, i+1, …, L if at beginning of year i have \$k. Note k integer $\leq M := wL(1+\rho)^L$

- $U[k,L] := \log(k)$
  Consumption = k, because at last year L he spends all

- $U[k,i] := \max_{c\,:\,0 \leq c \leq M} \log(c) + U[(k-c)(1+\rho)+w,\ i+1]$
  Consumption = argmax

- $\Rightarrow$ Dynamic programming algorithm running in time $O(LM^2)$

- Slightly more realism
- With probability q Bob earns interest rate (1+ρ)
- With probability 1-q Bob loses money rate (1- ρ)
- U[k,i] := expected utility for years i, i+1, …, L if at beginning of year i has $k
- U[k,L] := log(k)
- U[k,i] := $\max_{c:\, 0 \le c \le M}$ log (c) + ?

- Slightly more realism
- With probability q Bob earns interest rate $(1+\rho)$
- With probability 1-q Bob loses money rate $(1- \rho)$
- $U[k,i]$:= expected utility for years i, i+1, …, L if at beginning of year i has \$k
- $U[k,L] := \log(k)$
- $U[k,i] := \max_{c:\ 0 \leq c \leq M} \log(c) +$

  $q\, U[(k - c)(1+\rho) + w, i+1] + (1-q)\, U[(k - c)(1-\rho) + w, i+1]$

Subset sum problem

- Problem: Input integers w_1 , w_2 … , w_n , t
- Output: Number of (subsets) x ∈ $\{0,1\}^n$: $\sum_{i=1}^{n} w_i \cdot x_i = t$

Example:

n = 3, t = 12

w = {2, 3, 5, 7, 10}

t = 10+2, 7+5, 7+3+2

Output = 3

Subset sum problem

- Problem: Input integers w_1 , w_2 … , w_n , t
- Output: Number of (subsets) x $\in \{0,1\}^n$: $\sum_{i=1}^{n} w_i \cdot x_i = t$

Arriving at subproblems and recursion

To get to t we can either:

use w_n then need to get to t - w_n using w_1 , w_2 … , w_{n-1}

or not then need to get to t using w_1 , w_2 … , w_{n-1}

Subset sum problem

- Problem: Input integers w_1 , w_2 … , w_n , t
- Output: Number of (subsets) x ∈ $\{0,1\}^n$: $\sum_{i=1}^{n} w_i \cdot x_i = t$

- Subproblems and recursion:
- S(i,s) := number of x ∈ $\{0,1\}^i$ such that $\sum_{j=1}^{i} w_j \cdot x_j = s$
- Recursion: S(i,s) = S(i-1,s) + S(i-1,s–w_i)

- There are only tn different subproblems S(i,s)

  (Don't need to consider sums larger than t)

  NOTE: Assuming weights are positive: $w_i \geq 0$ for all i

- Problem: Input integers w_1 , w_2 … , w_n , t
- Output: Number of (subsets) $x \in \{0,1\}^n : \sum_{i=1}^{n} w_i \cdot x_i = t$

Sum s

	1				
			,,,		
1	2				
1	1				

i = 1...n

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

  (for s = 0 … t)

  ?

Algorithm

- Problem: Input integers w_1 , w_2 … , w_n , t
- Output: Number of (subsets) $x \in \{0,1\}^n$: $\sum_{i=1}^{n} w_i \cdot x_i = t$

Sum s

	1				
			,,,		
1	2				
1	1				

i = 1...n

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

  (for s = 0 … t)

  $S(i,s) = S(i-1,s) + S(i-1,s–w_i)$

Algorithm

- T(n) = ?

- Problem: Input integers w_1 , w_2 … , w_n , t
- Output: Number of (subsets) $x \in \{0,1\}^n : \sum_{i=1}^{n} w_i \cdot x_i = t$

Sum s

	1				
			,,,		
1	2				
1	1				

i = 1...n

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

  (for s = 0 … t)

  $S(i,s) = S(i-1,s) + S(i-1,s-w_i)$

- T(n) = O(tn)

- Problem: Input integers w_1 , w_2 … , w_n , t
- Output: Number of (subsets) x ∈ $\{0,1\}^n$: $\sum_{i=1}^{n} w_i \cdot x_i = t$

Sum s

	1				
			,,,		
1	2				
1	1				

i = 1...n

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

  (for s = 0 … t)

  $S(i,s) = S(i-1,s) + S(i-1,s-w_i)$

- Space: Trivial: O(tn) Better: ??

- Problem: Input integers w_1 , w_2 … , w_n , t
- Output: Number of (subsets) $x \in \{0,1\}^n$: $\sum_{i=1}^{n} w_i \cdot x_i = t$

Sum s

	1				
			,,,		
1	2				
1	1				

i = 1...n

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

  (for s = 0 … t)

  $S(i,s) = S(i\text{-}1,s) + S(i\text{-}1,s\text{–}w_i)$

- Space: O(t), just keep two columns

Example:

$n = 3$, $t = 12$

$w = \{2, 3, 5, 7, 10\}$

$t = 10+2,\ 7+5,\ 7+3+2$

Output = 3

12				2	3
11					
10			1	2	3
9				1	1
8			1	1	1
7			1	2	2
6					
5		1	2	2	2
4					
3		1	1	1	1
2	1	1	1	1	1
1					
0	1	1	1	1	1
	1	2	3	4	5

Greedy Algorithms

Dynamic programming requires solving all subproblems, leads to algorithms with running time usually n^2 or n^3

Sometimes, greedy is faster.

A greedy algorithm always makes the choice that looks best at the moment.

That is, it keeps making locally optimal decision in the hope that this will lead to a globally optimal solution.

Activity Selection problem

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

$$A \coloneqq \{a_1, a_2, \dots a_n\}$$

Activity a_i takes time $[s_i, f_i)$.

Activities a_i, a_j are compatible if $s_j \geq f_i$

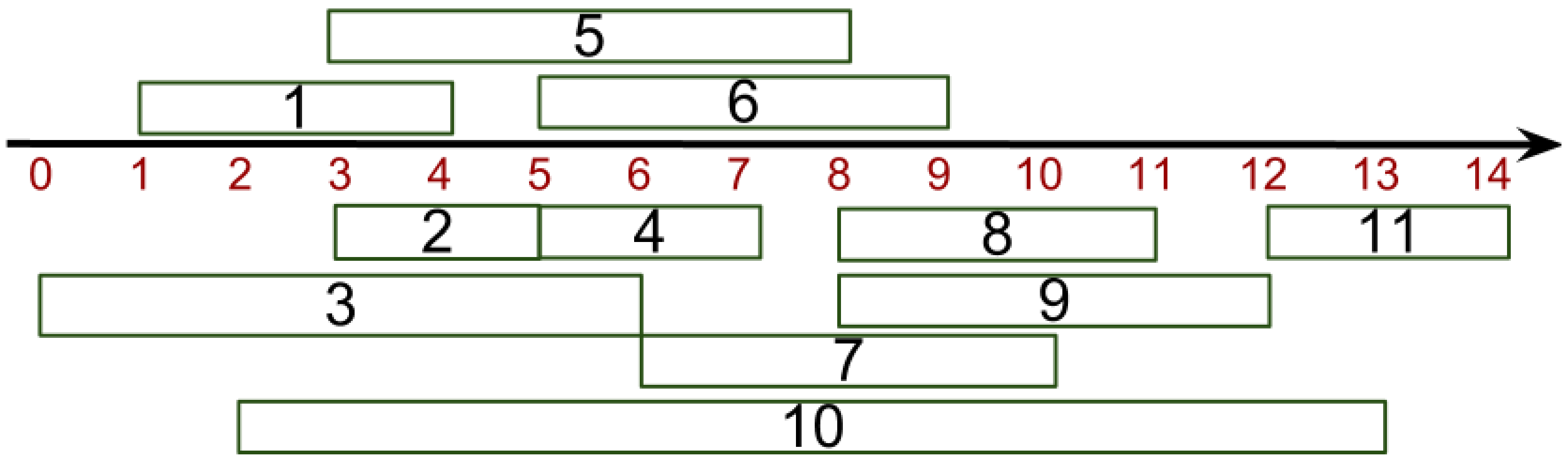
Output:

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

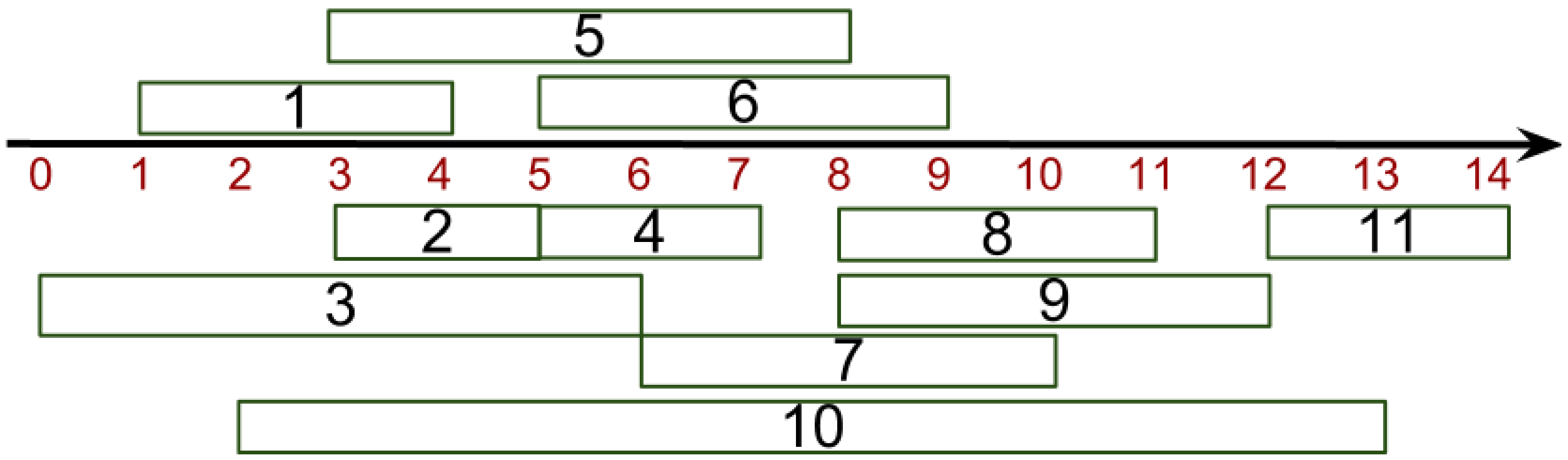
A set of compatible activities = ?



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}).

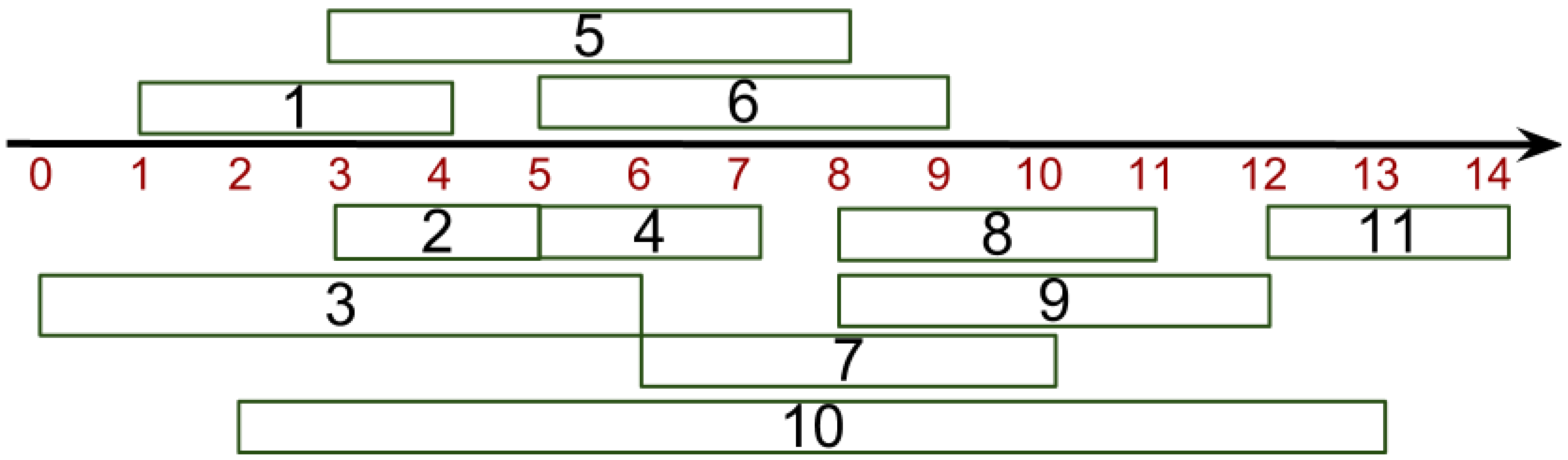


Example:

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 = ?

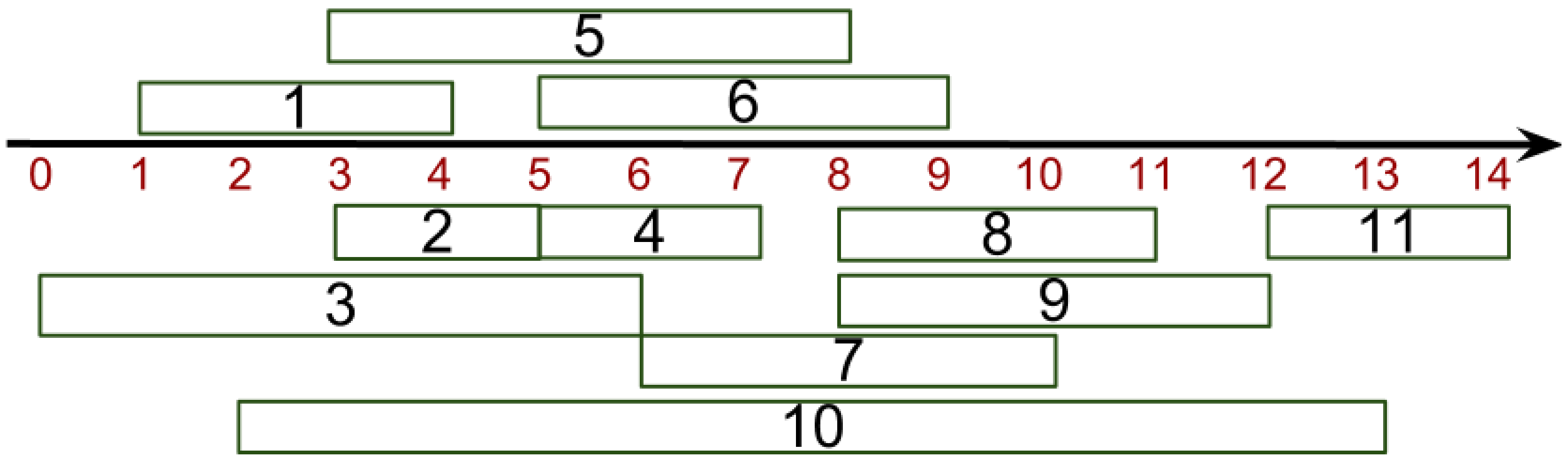


Example:

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 = (a1,a4,a8,a11)



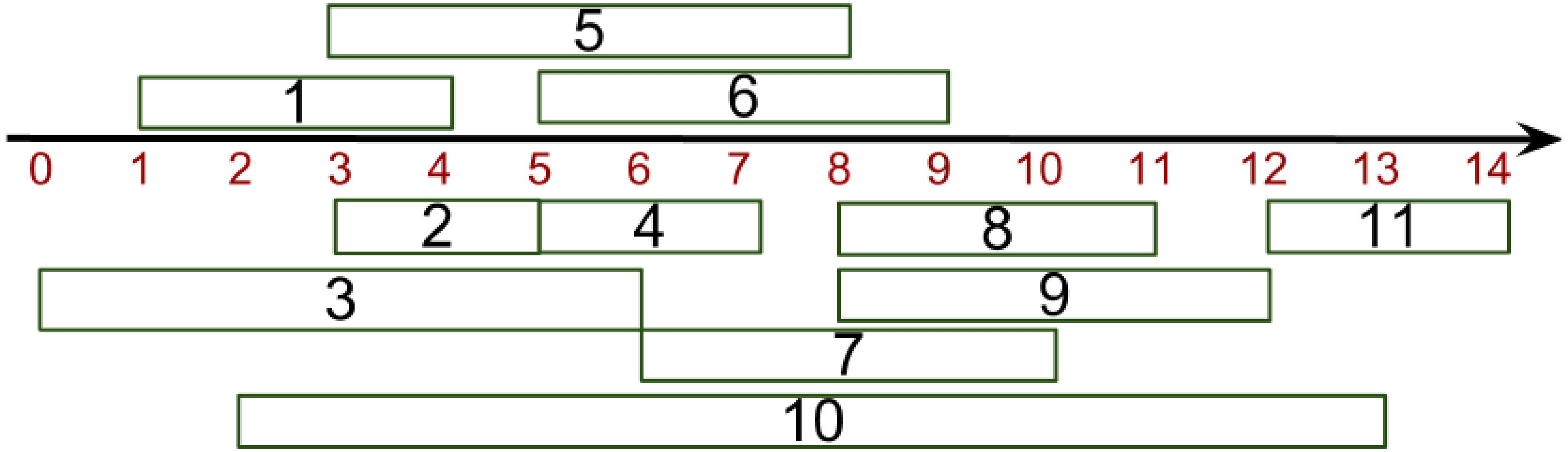
Example:

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 = (a1,a4,a8,a11)

Is there another maximal set ?



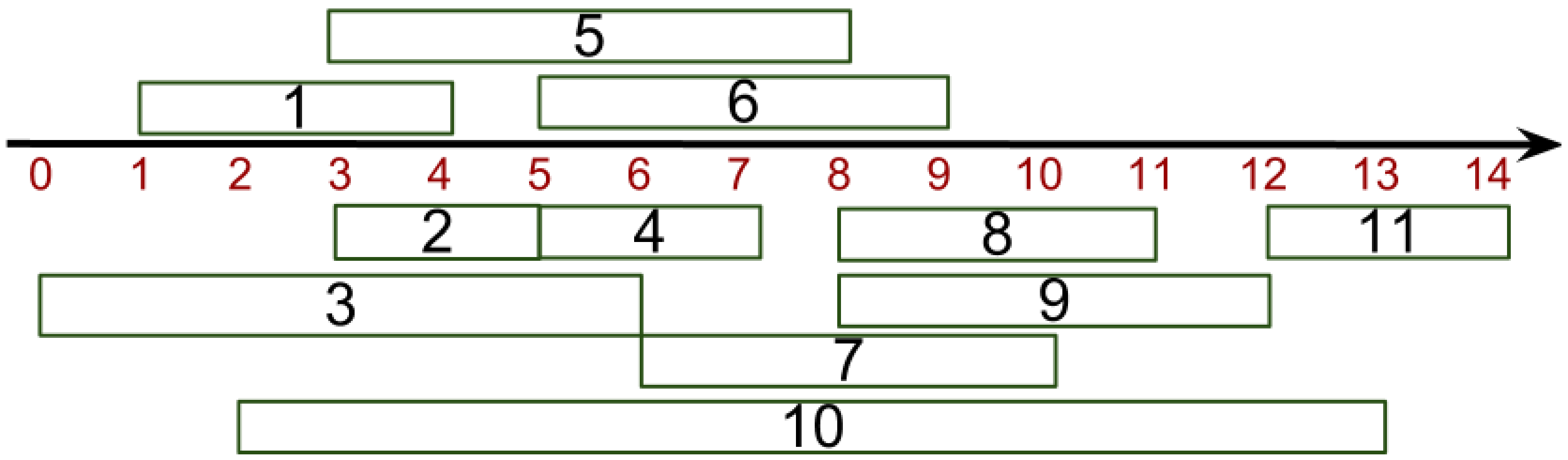
Example:

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 = (a1,a4,a8,a11)

Is there another maximal set? Yes. (a2,a4,a9,a11)



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

- Proof:

  Let [s*,f*) be activity with earliest finish time f*

  Let S be an optimal solution

  Write S = S' U [s,f) where [s,f) has earliest finish time among activities in S

- Then S' U [s*,f*) is also an optimal solution, because every activity in S' has start time > f > f*.

- Greedy Algorithm:

  Pick activity with earliest finish time,

  that does not overlap with activities already picked

  Repeat

- Claim: The algorithm is correct
- Proof: Follows from applying previous claim iteratively.

- Let us see the algorithm in more detail

Greedy activity selection algorithm

```
activity-selection(A) {
sort A increasingly according to f [i];
n:= length[A];
S:=a[1]
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[1]
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	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

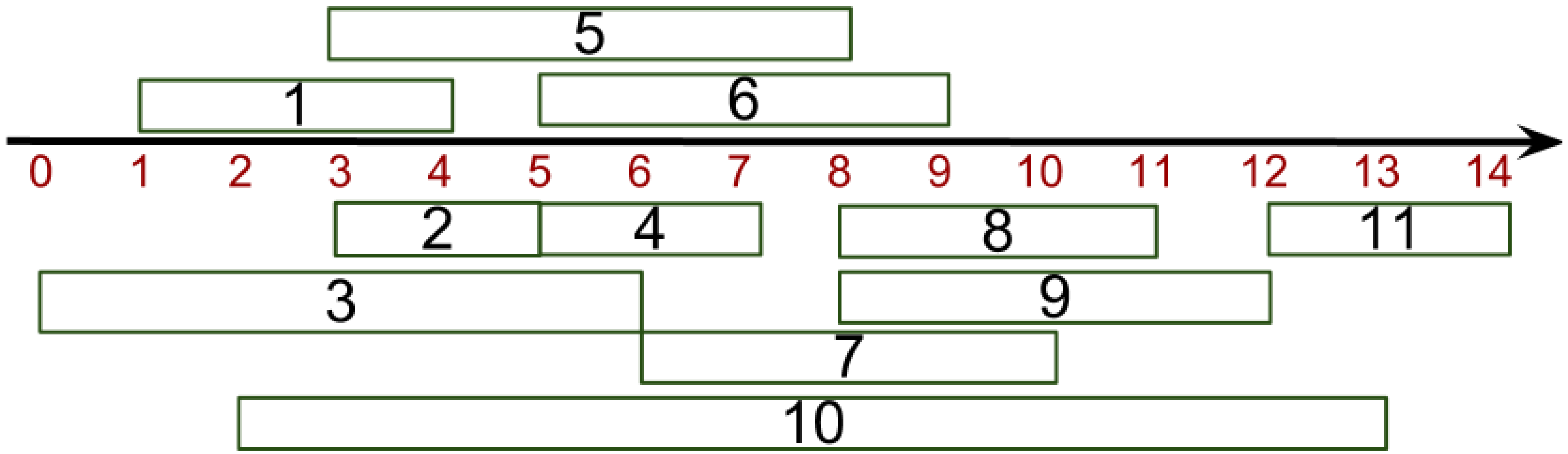
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:= a[1]
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

Already sorted

according to finish time.

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

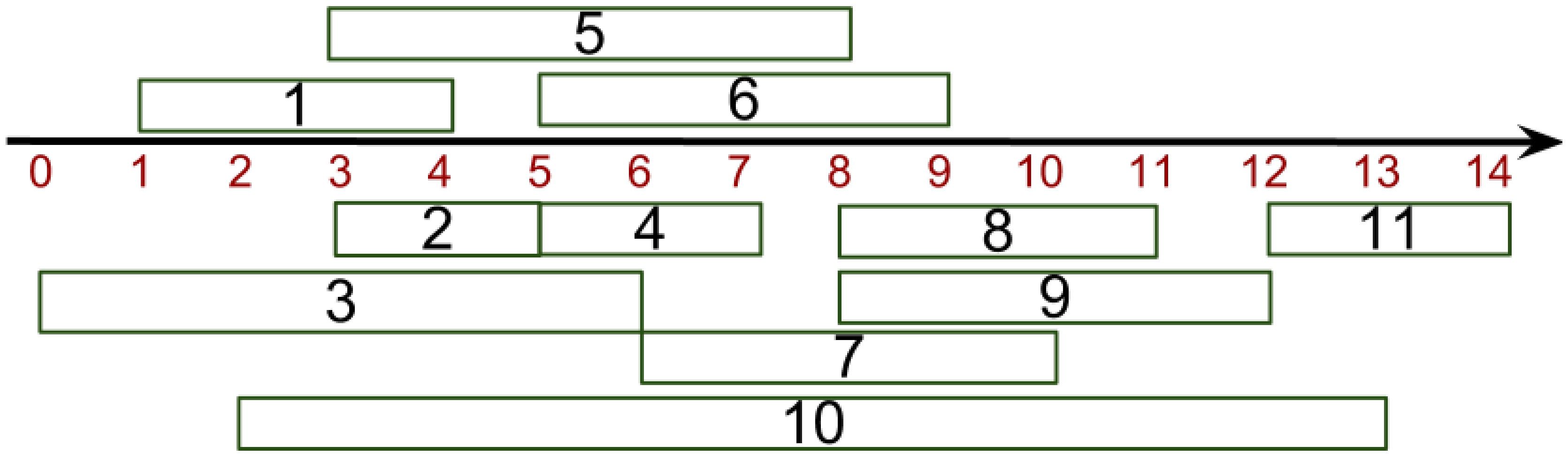


```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:= a[1]
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

n:=11

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



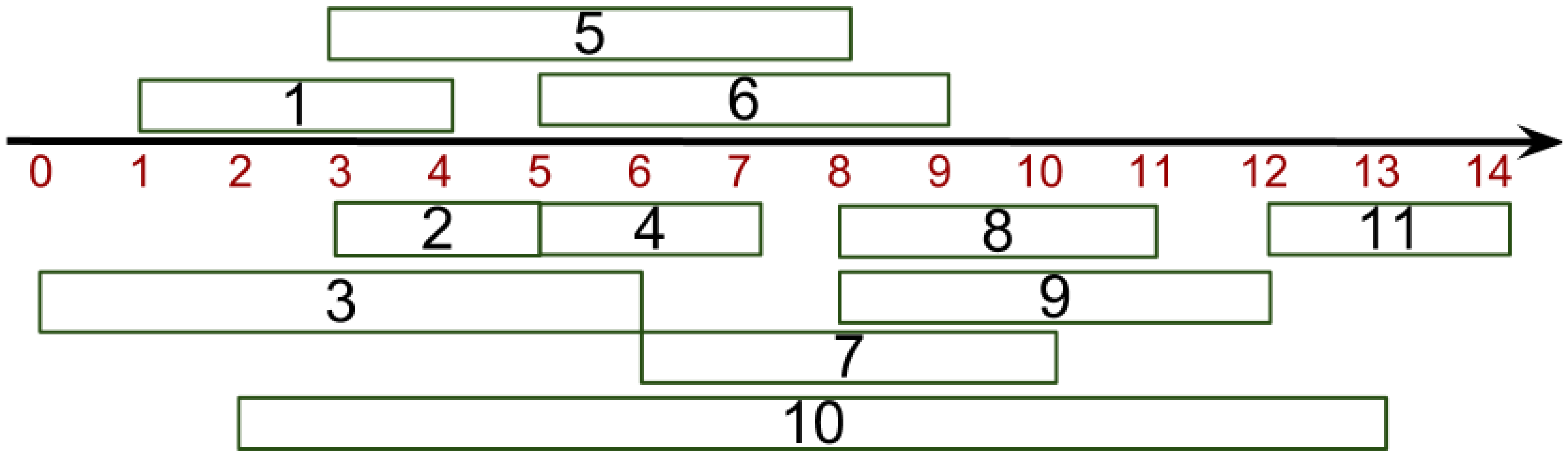
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:= a[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

$S:=\{a_1\}$

n:=11

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

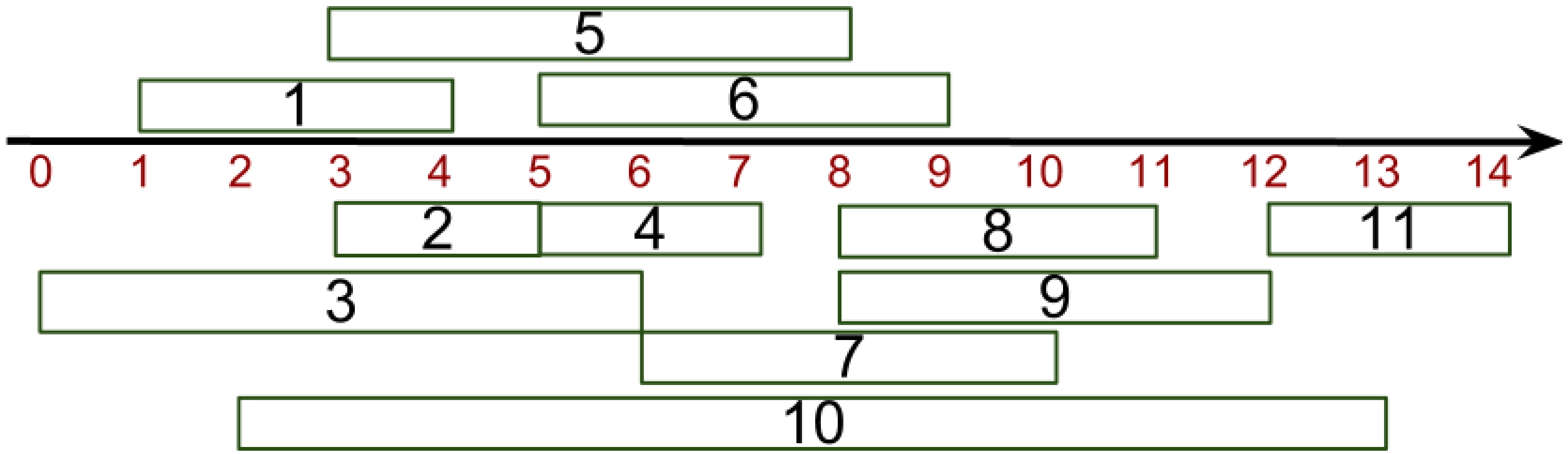


```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:= a[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

$S := \{a_1\}$

a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

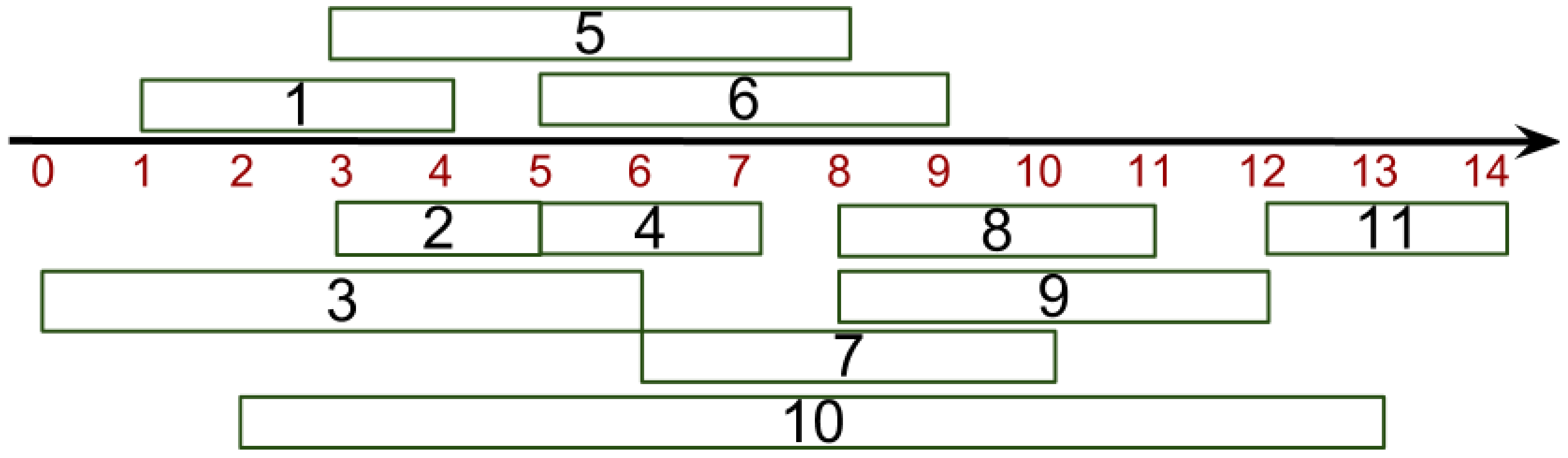
Example:

$S:=\{a_1\}$

i m

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

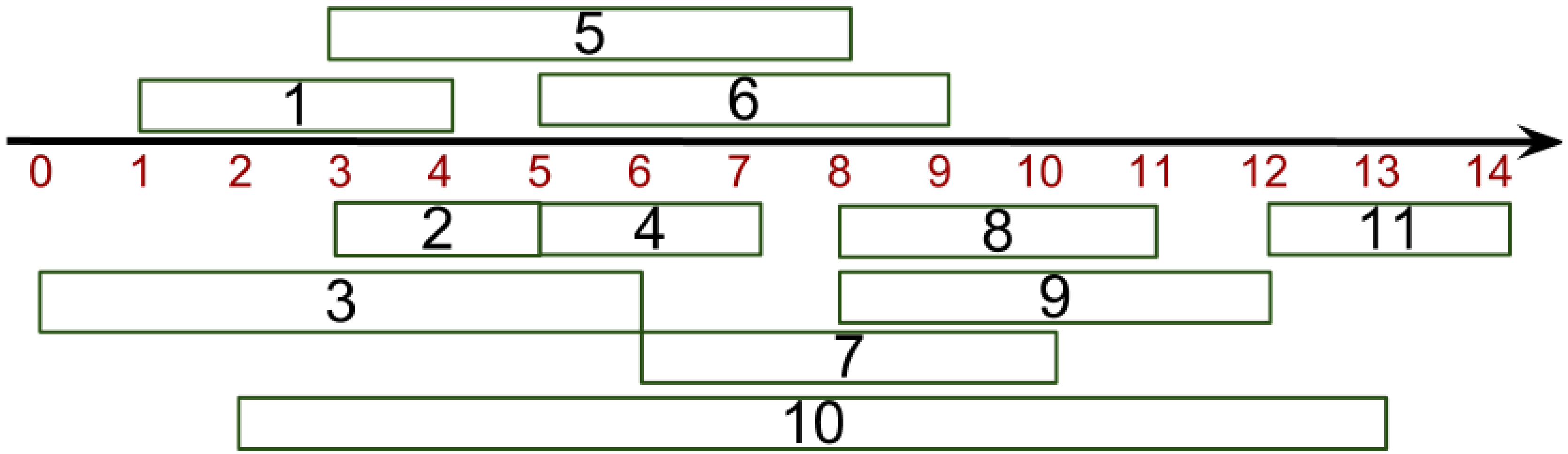
$S:=\{a_1\}$

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

i m

n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
         Add a[i] to S;
         i :=m;} return S;
}
```

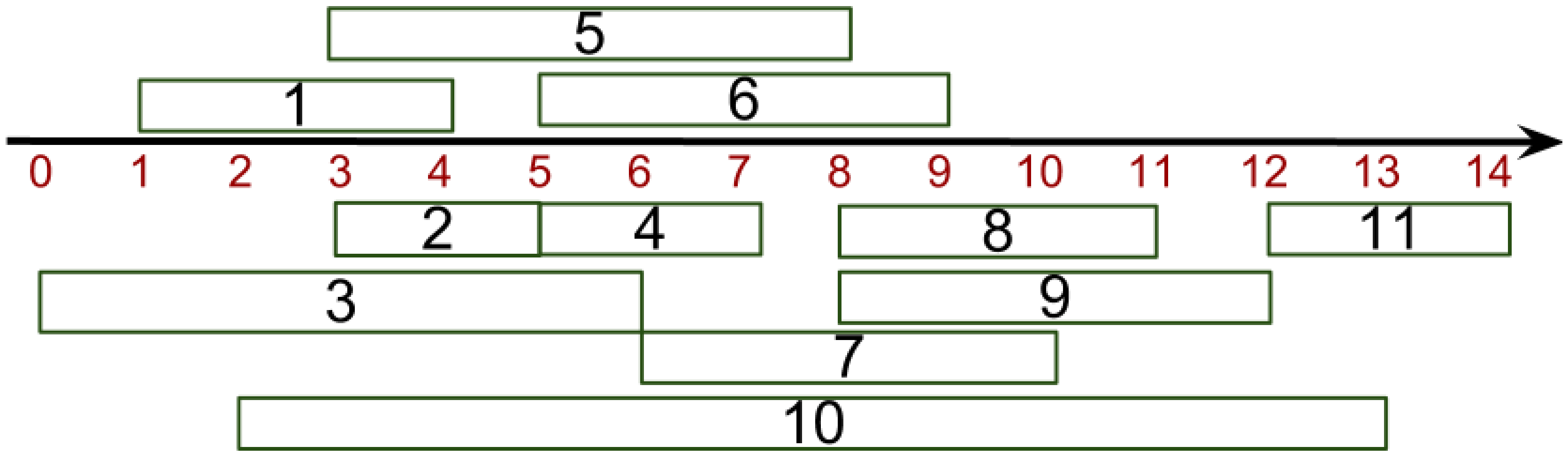
Example:

$S:=\{a_1\}$

$s[2] < f[1]$

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

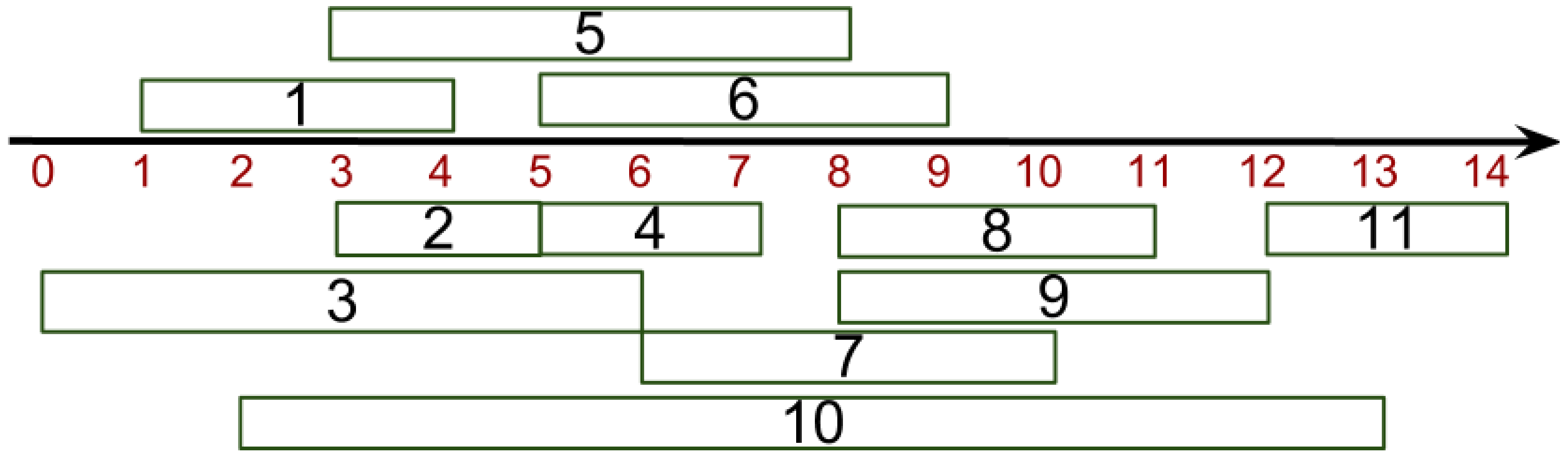
Example:

$S:=\{a_1\}$

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

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

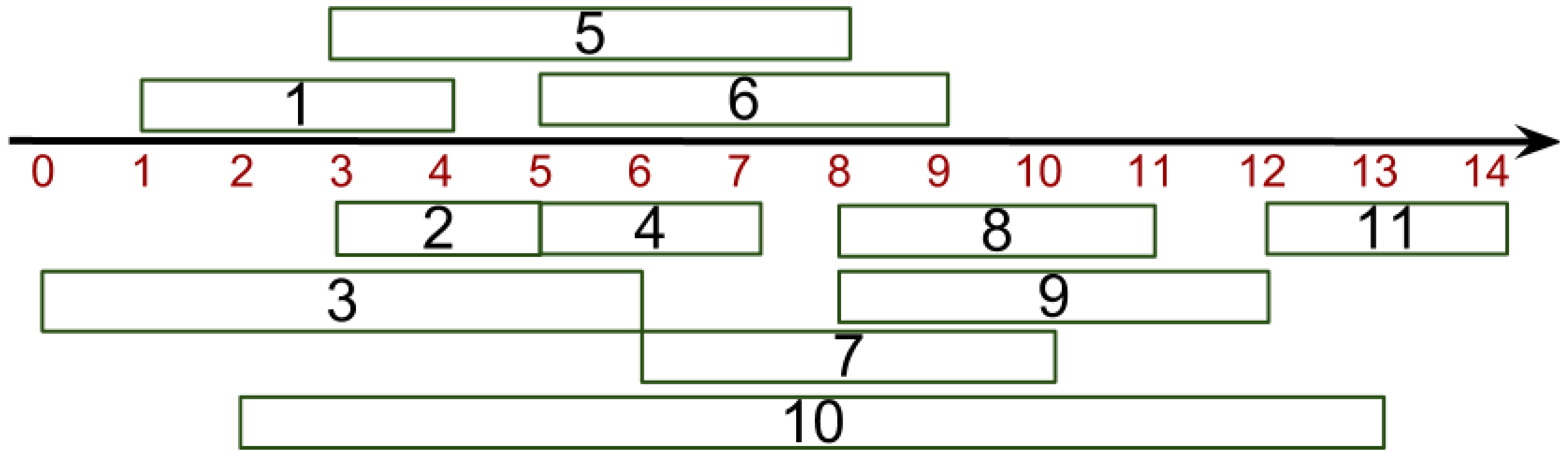
Example:

$S:=\{a_1\}$

$s[3] < f[1]$

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

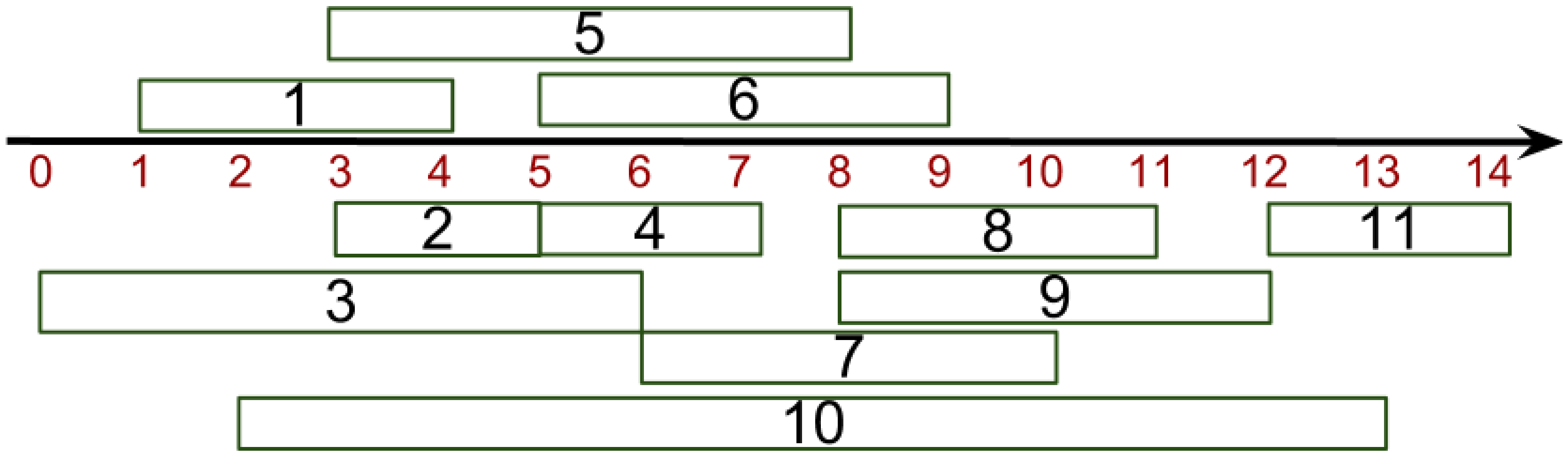
Example:

$S:=\{a_1\}$

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

i m n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
         Add a[i] to S;
         i :=m;} return S;
}
```

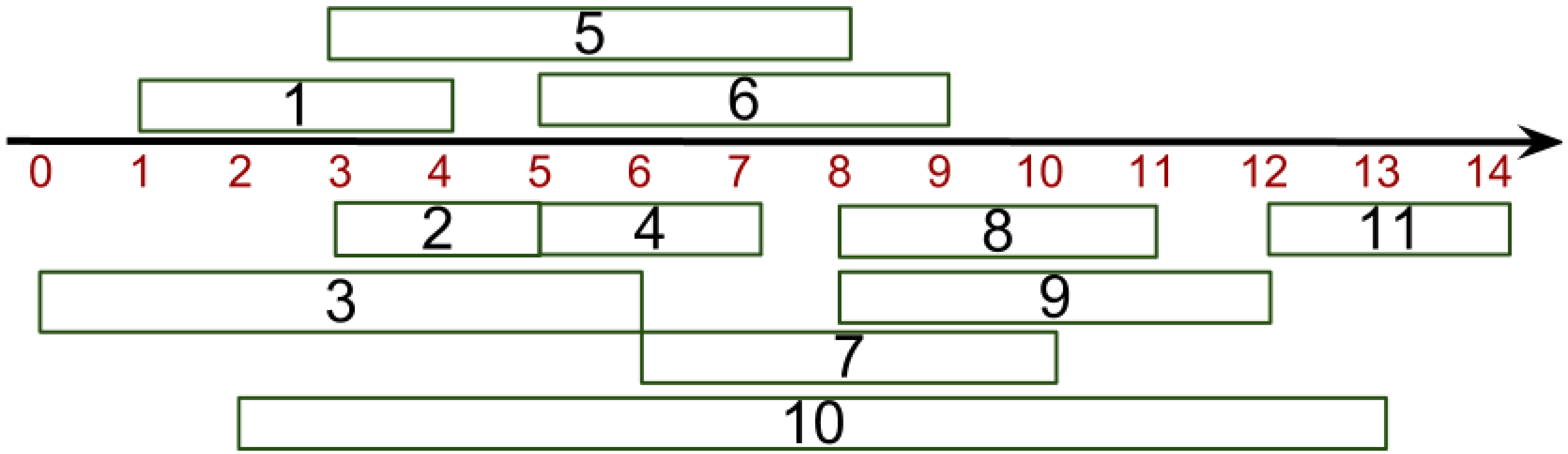
Example:

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

$s[4] > f[1]$

i   m   n:=11

a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

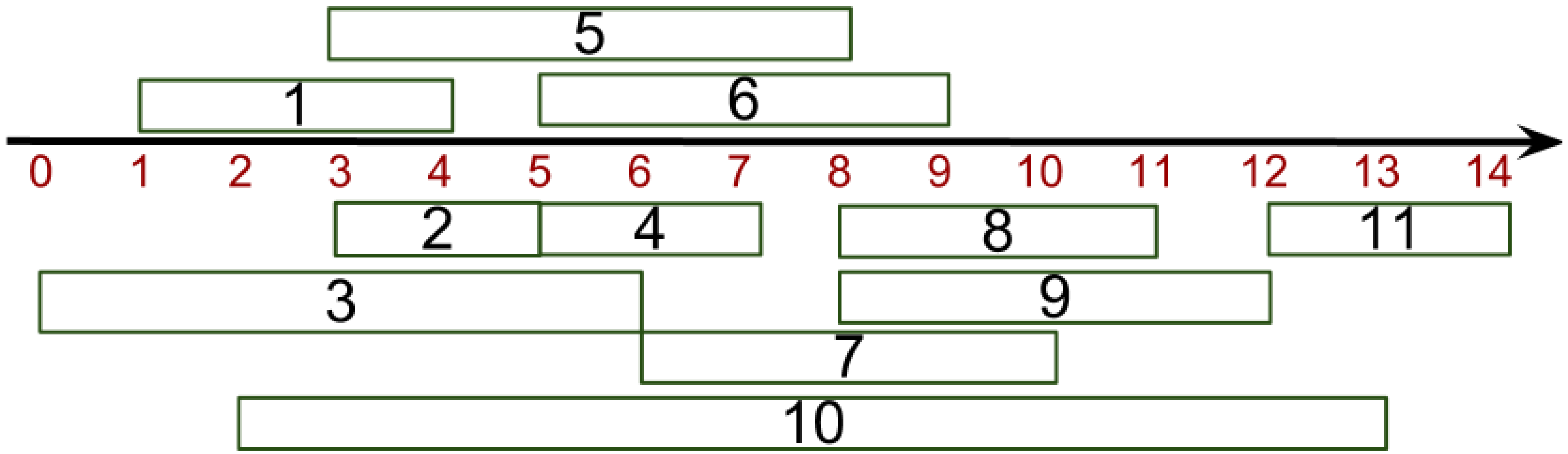
$S:=\{a_1 ,a_4\}$

$s[4] > f[1]$

i,m

n:=11

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



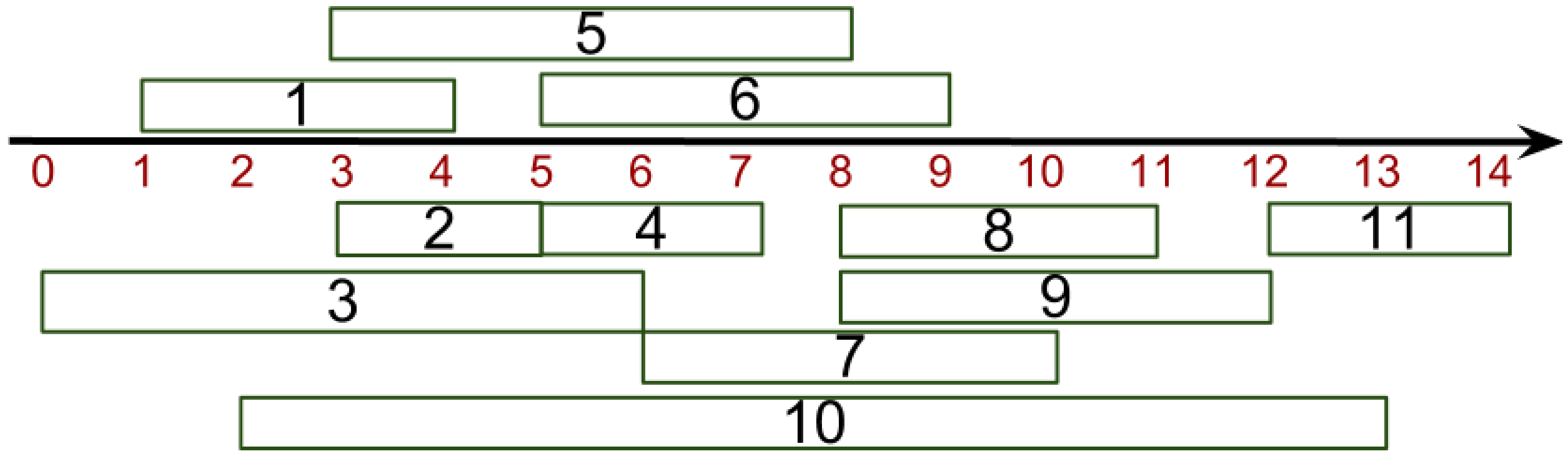
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:= a[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

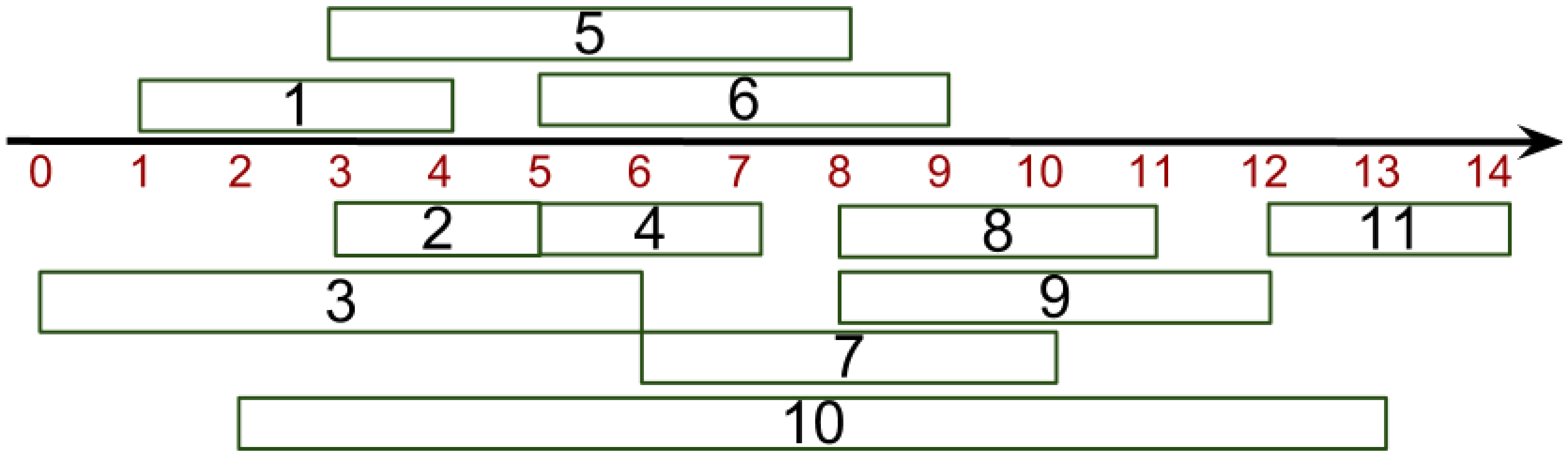
Example:

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

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

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[1]
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

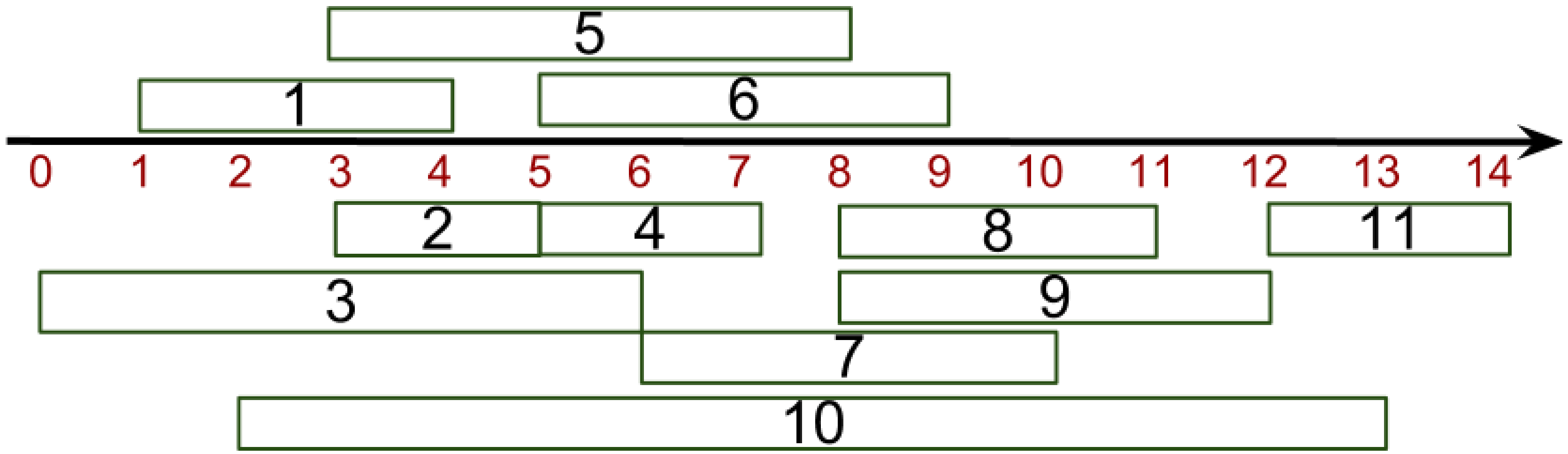
Example:

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

$s[5] < f[4]$

i 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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

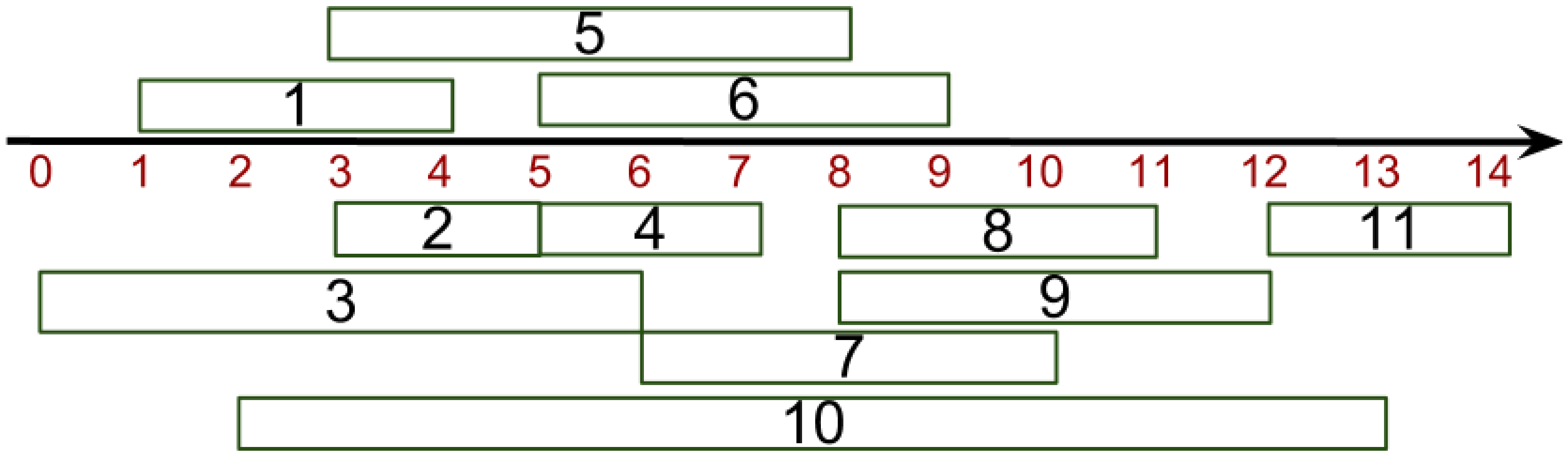
Example:

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

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

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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

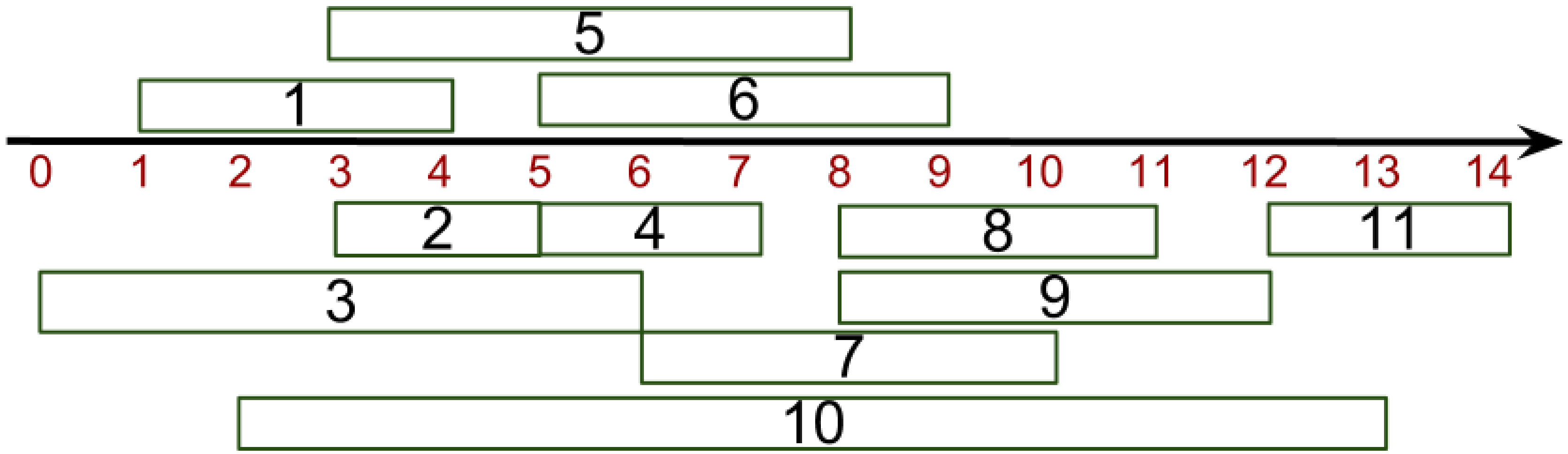
Example:

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

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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

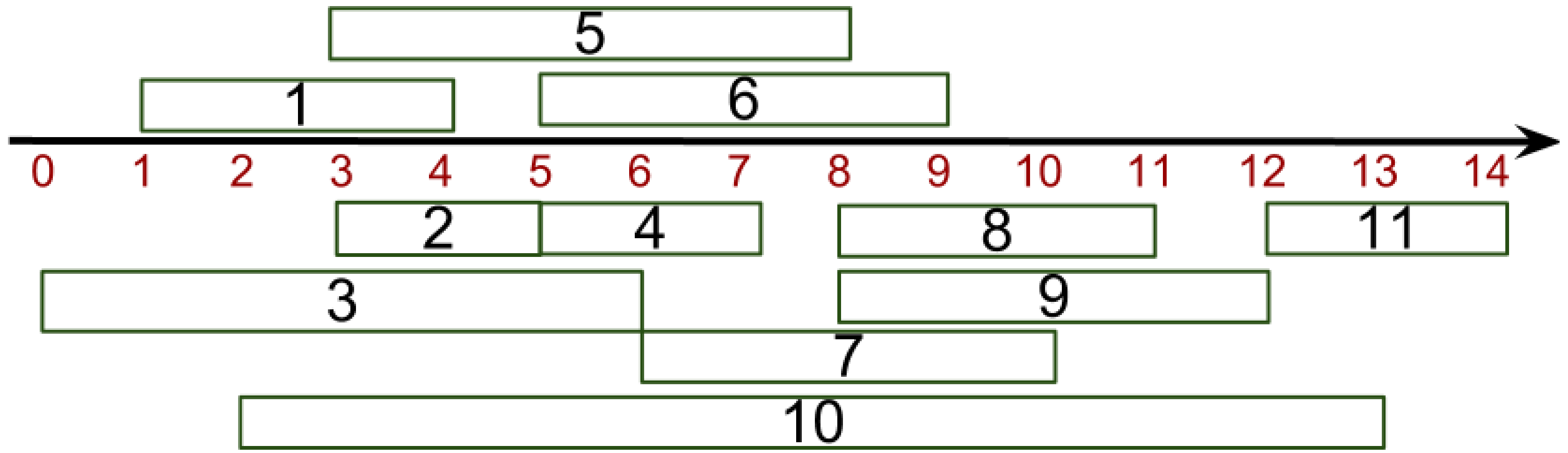
Example:

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

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

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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

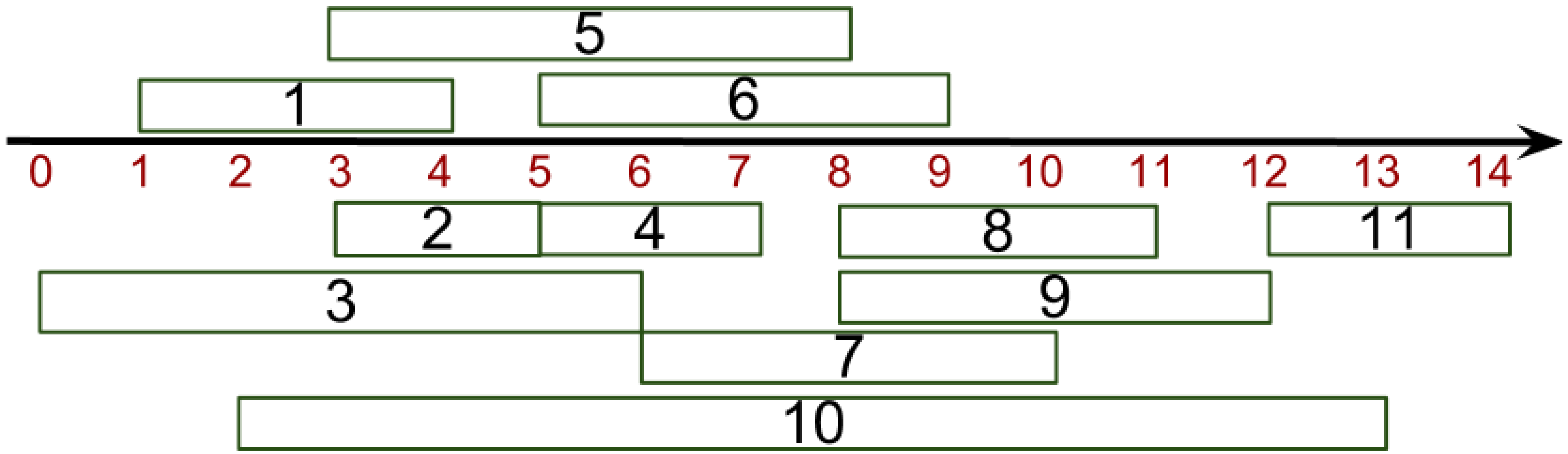
Example:

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

$s[7] < f[4]$

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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

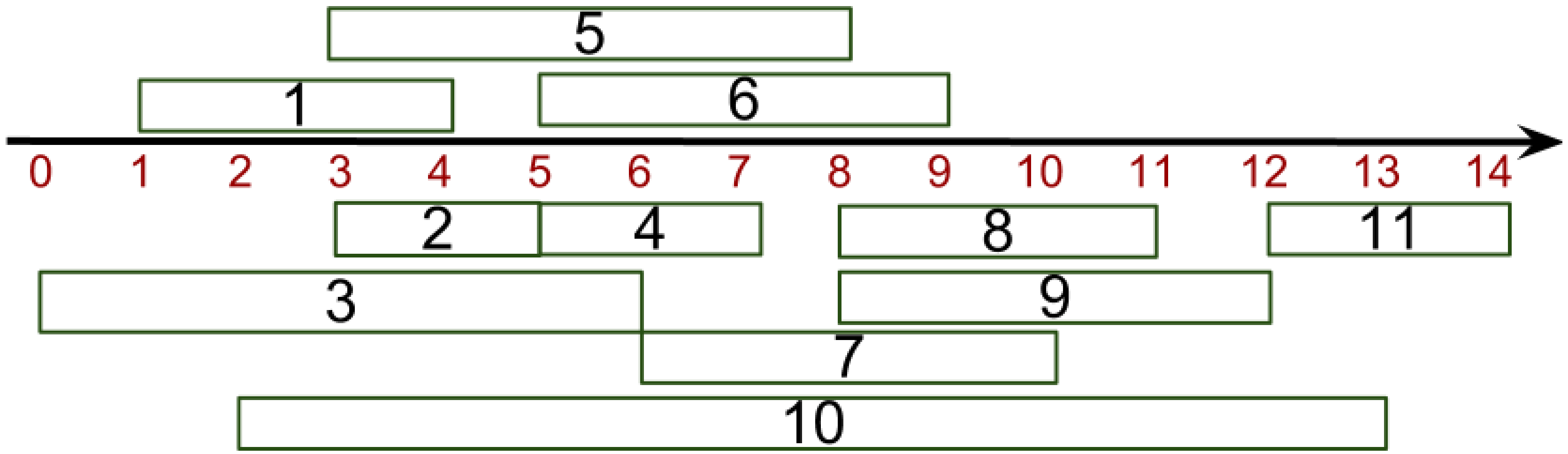
Example:

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

$s[8] \geq f[4]$?

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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

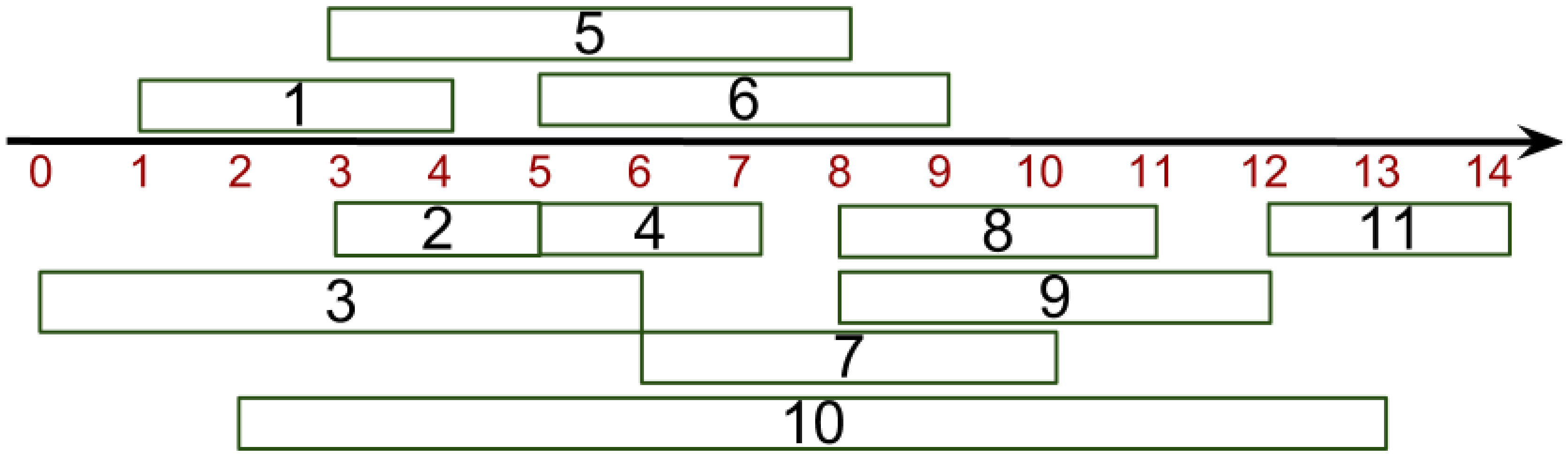
Example:

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

$s[8] > f[4]$

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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

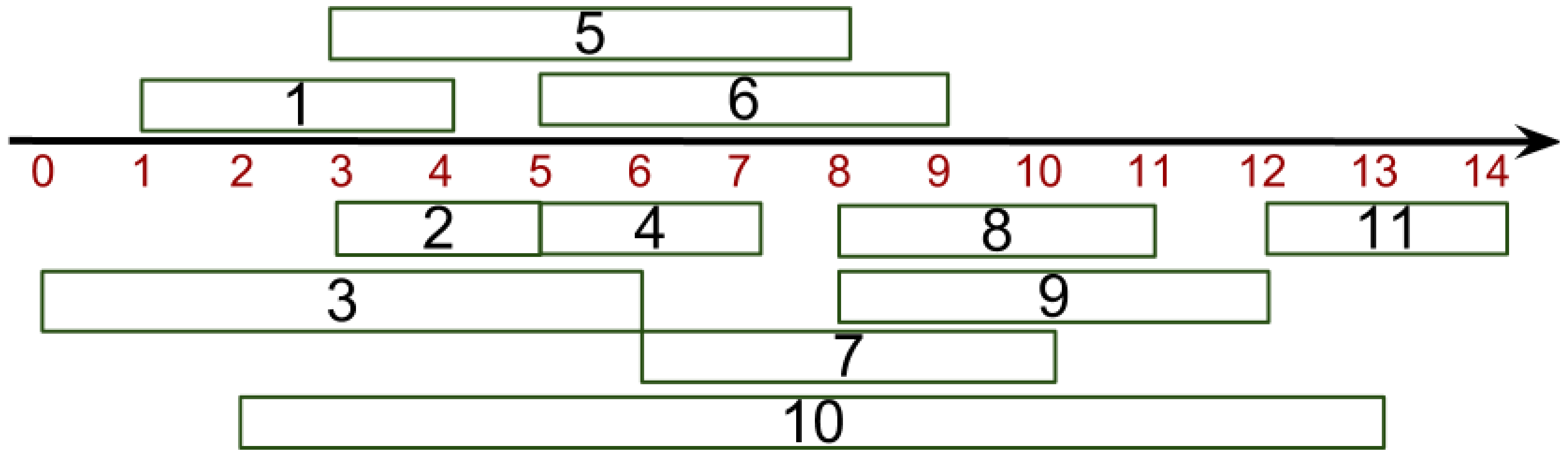
Example:

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

s[8] > f[4]

i,m n:=11

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



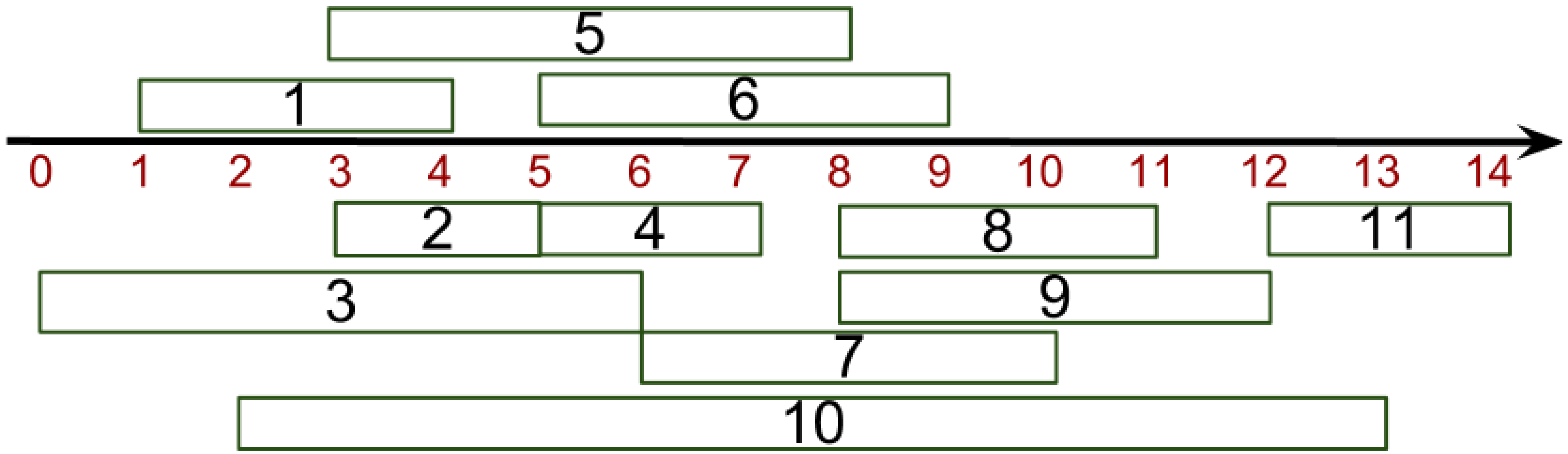
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:= a[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

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	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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

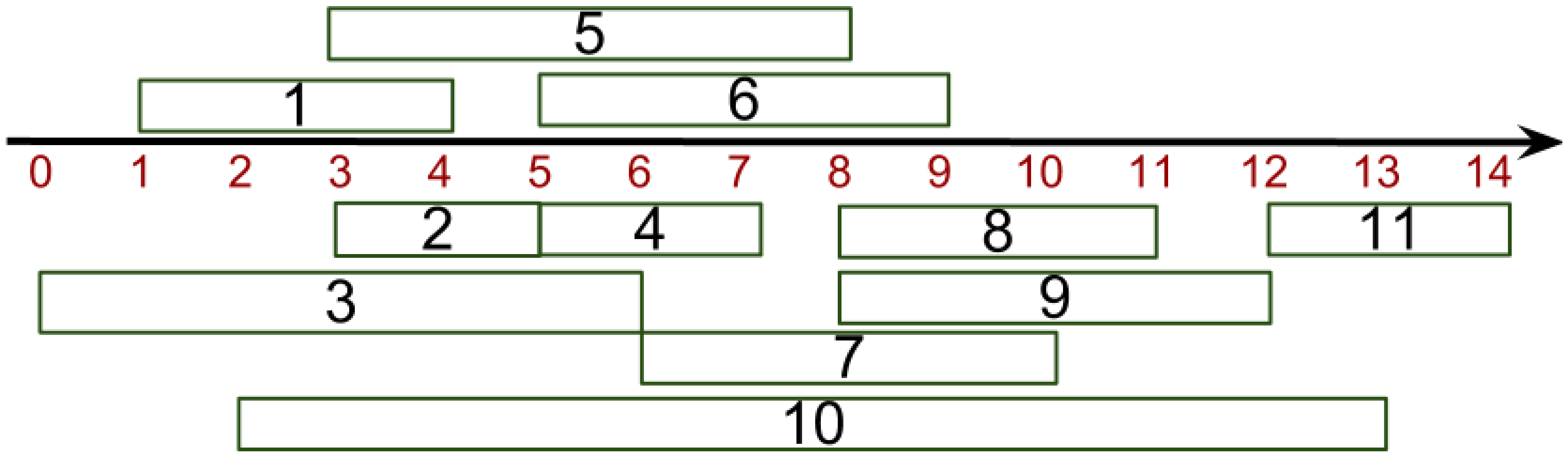
Example:

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

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

i m n:=11

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



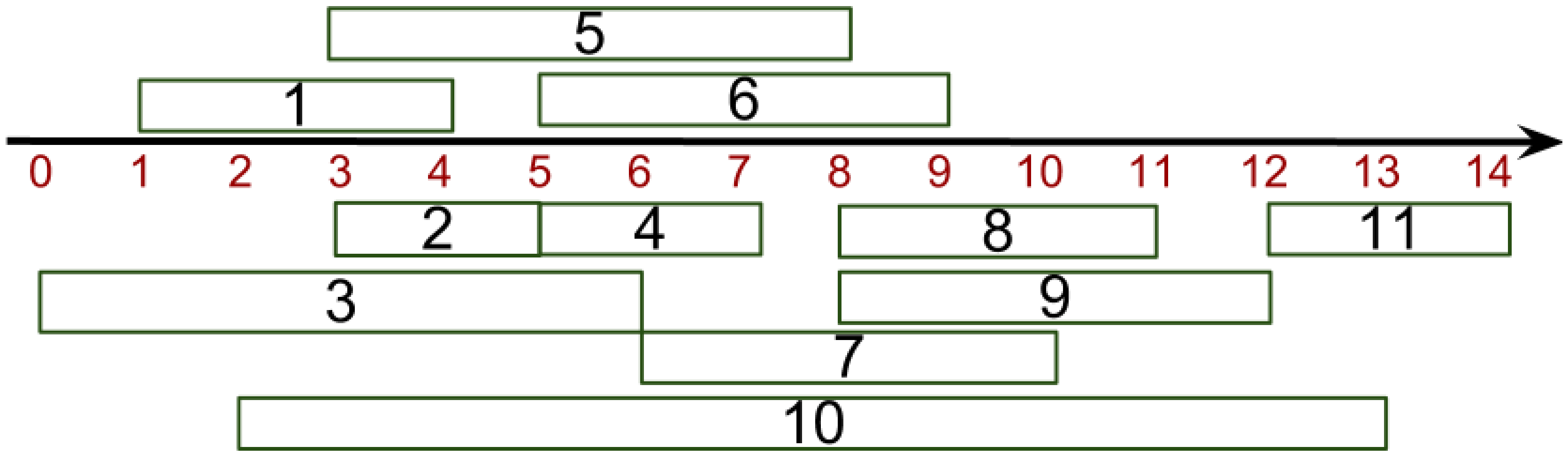
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:= a[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

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



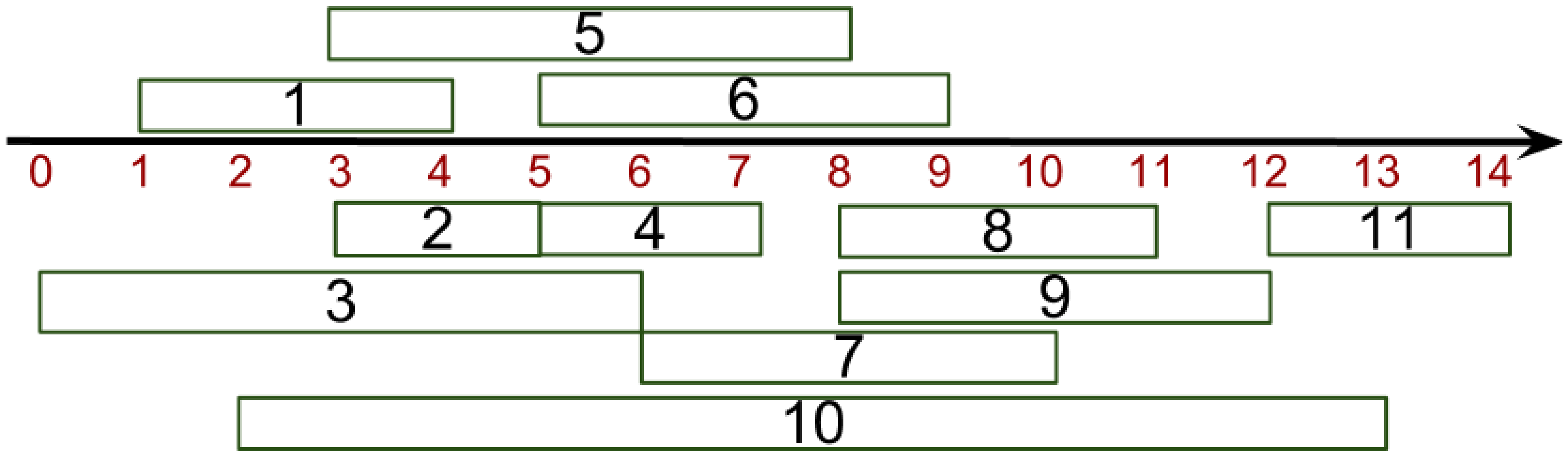
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:= a[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

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



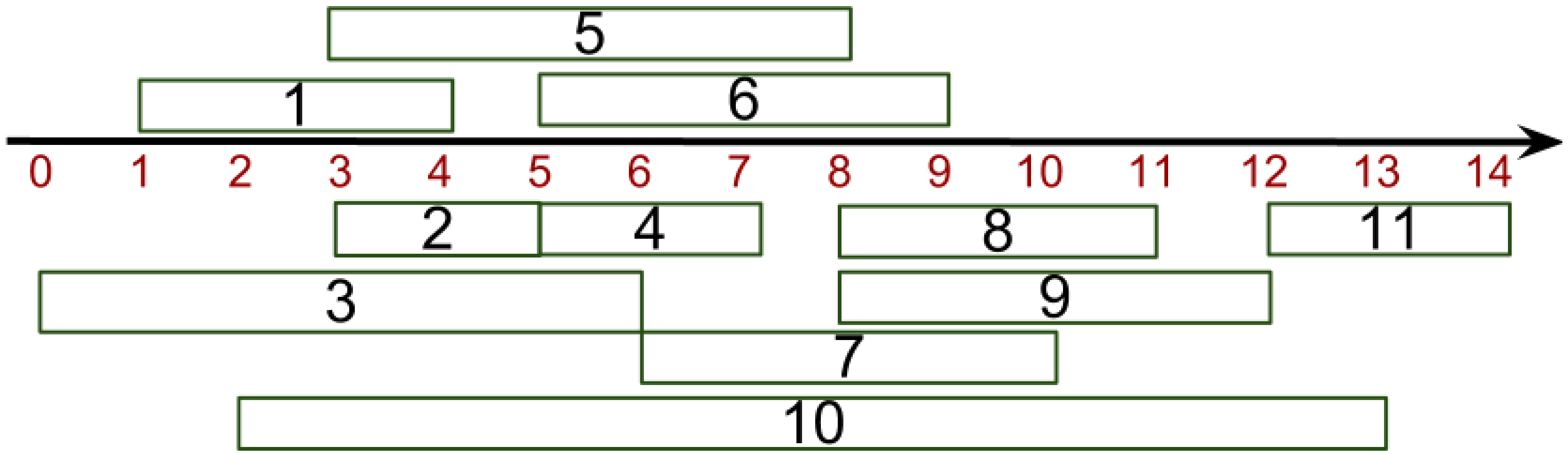
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:= a[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

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



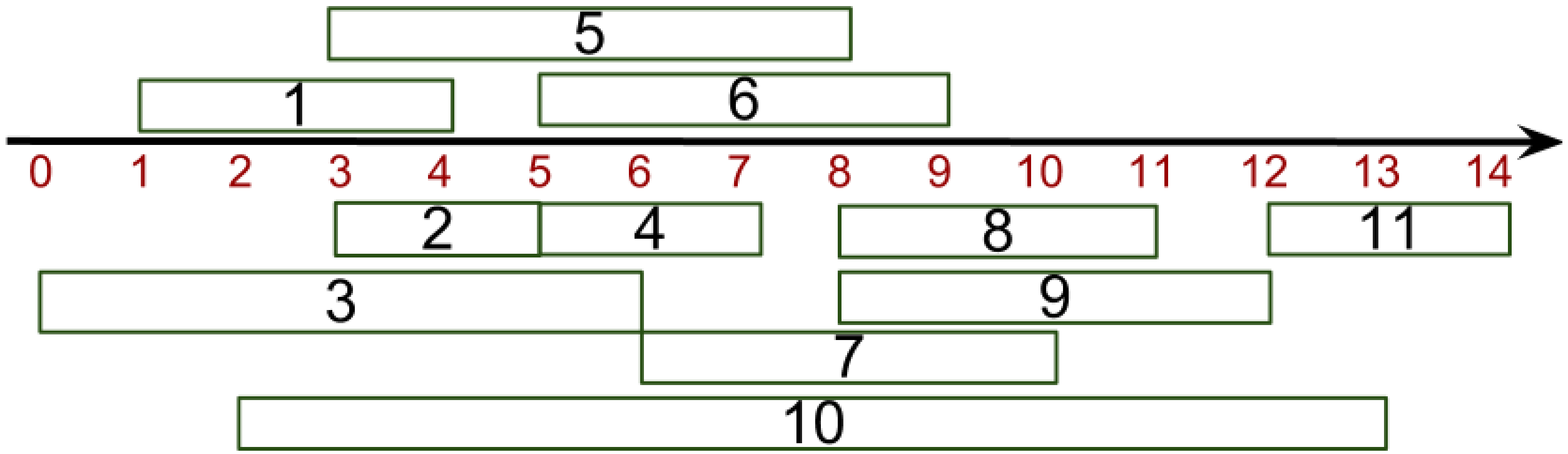
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:= a[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
       Add a[i] to S;
       i :=m;} return S;
}
```

Example:

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

$S[11] \geq f[8]$?

								i			m
a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

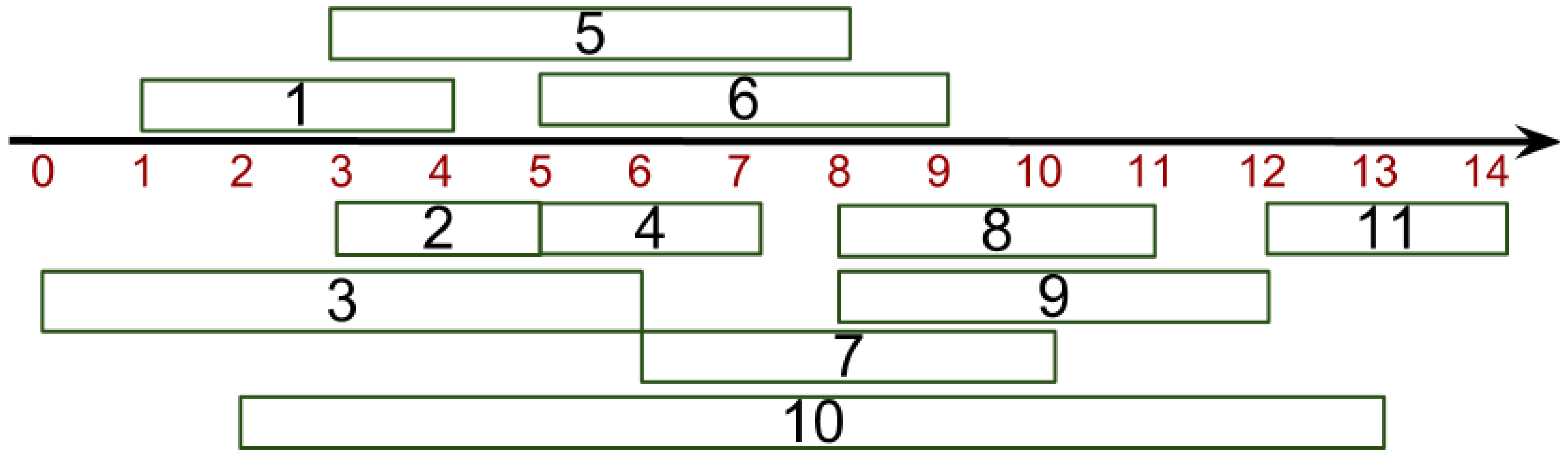
Example:

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

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

i m

a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

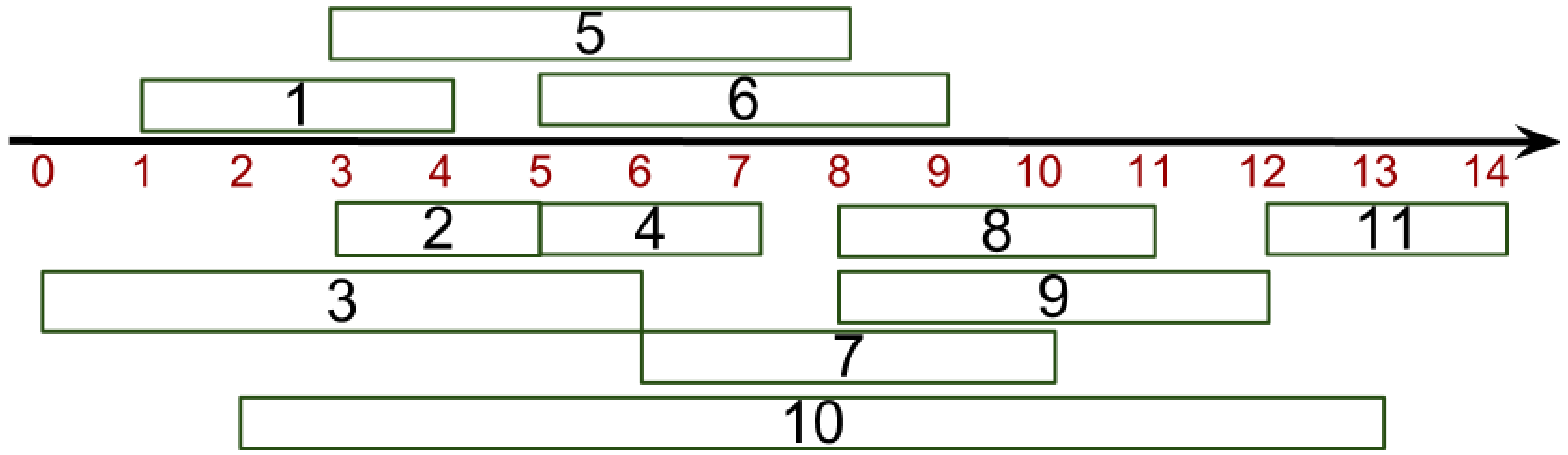
Example:

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

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

i, m

a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
         Add a[i] to S;
         i :=m;} return S;
}
```

Example:

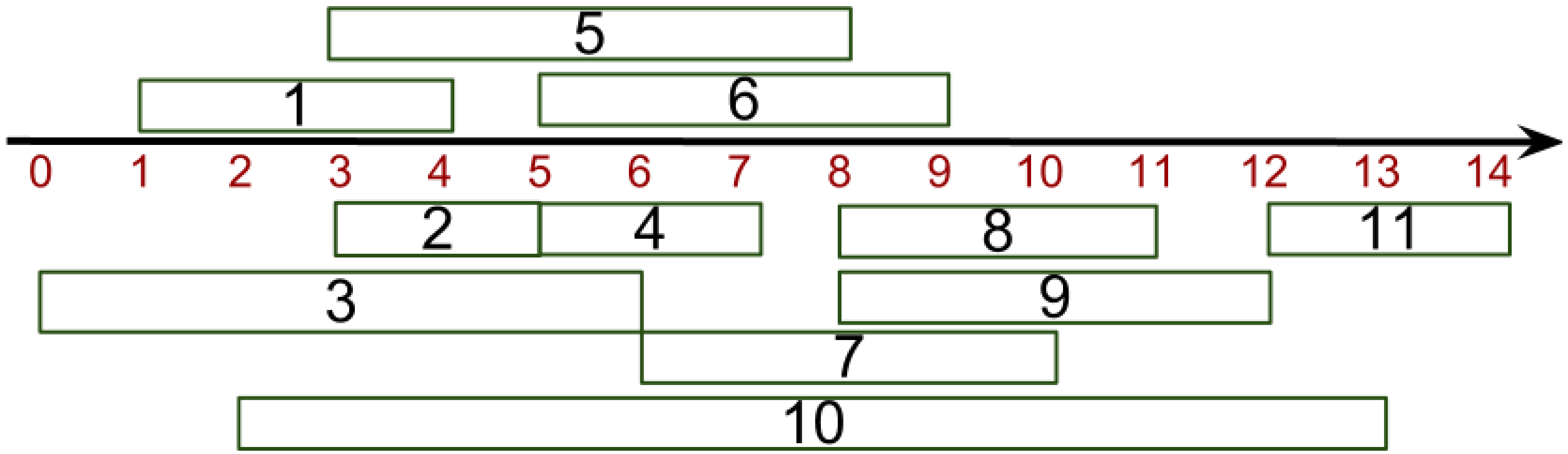
$S:=\{a_1, a_4, a_8, a_{11}\}$

m = 12,

n = 11

i

a_i	1	2	3	4	5	6	7	8	9	10	11
s	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[1]
i:=1;
for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

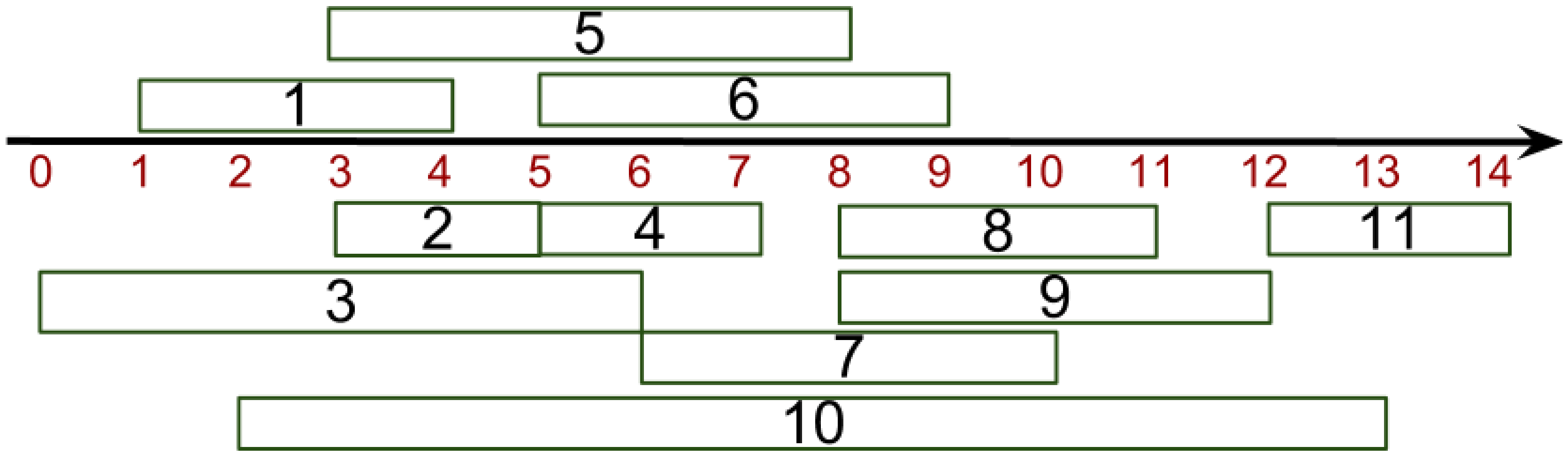
$S:=\{a_1, a_4, a_8, a_{11}\}$

m = 12,

n = 11

i

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

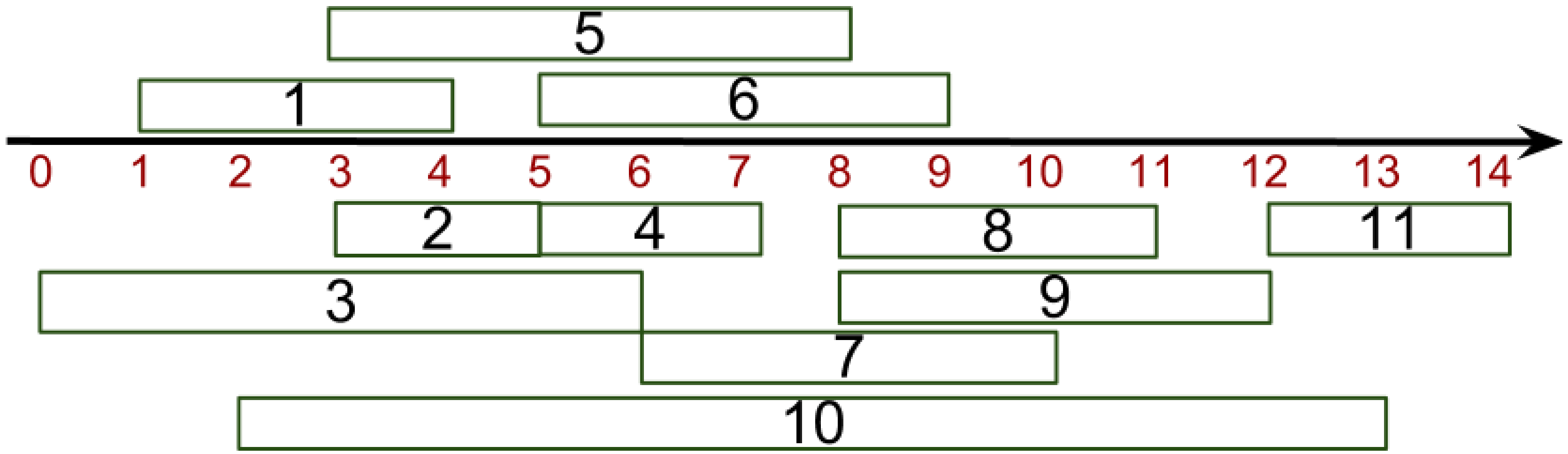


```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:= a[1]
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
        Add a[i] to S;
        i :=m;} return S;
}
```

Example:

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

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



Deleted scenes

Knapsack

Input:

- S:={(v_1 ,w_1), (v_2 ,w_2), … (v_n ,w_n)}.

  (v_i , w_i) means item i is worth v_i and weighs w_i .

- W, weight-capacity of knapsack.

Output:

- Items that maximize value in knapsack.

Can we take a fraction of an item?

Fractional Knapsack : Yes

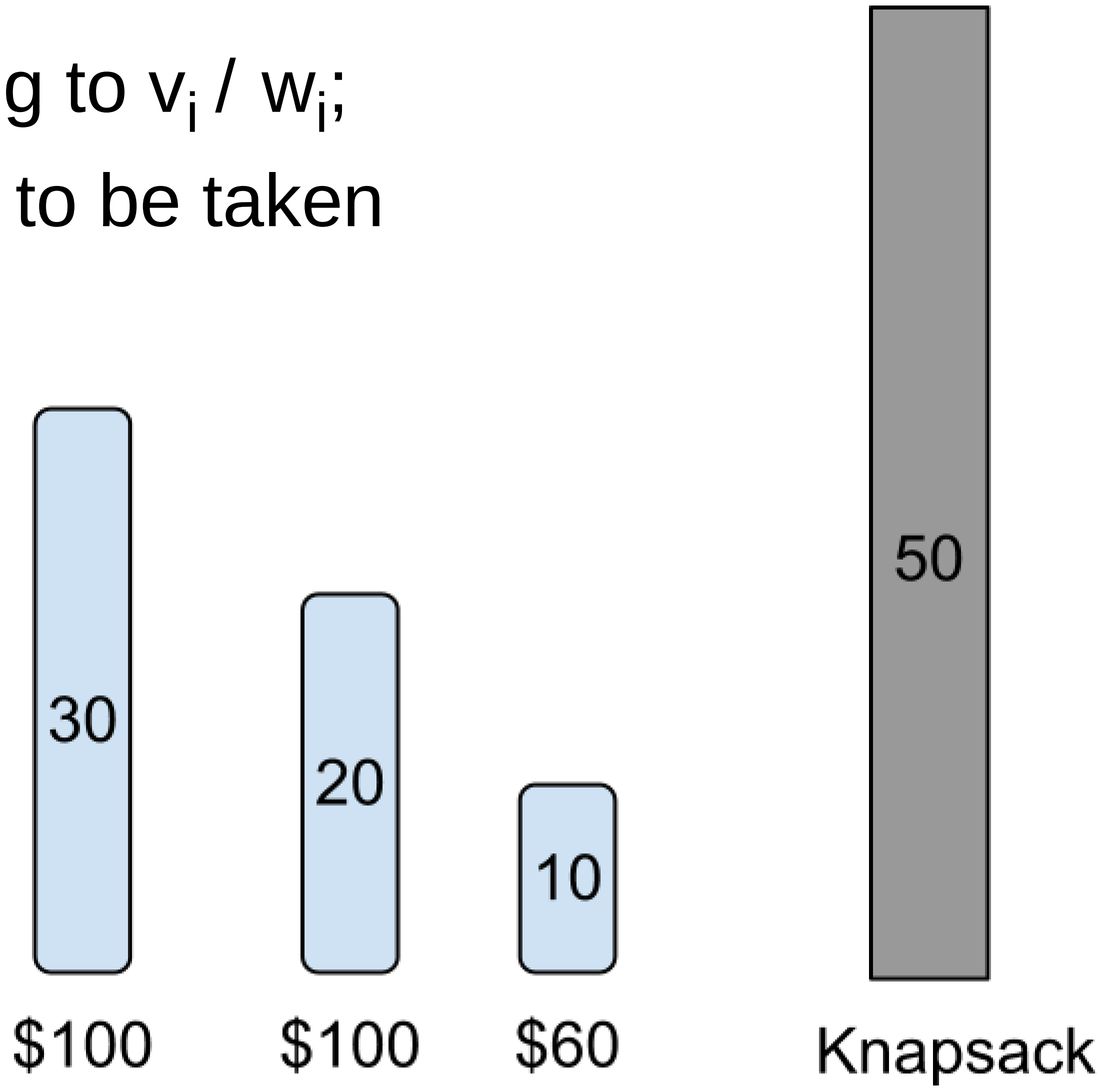
0-1 Knapsack : No

Fractional Knapsack

- Compute v_i / w_i for each item.
- Sort S according to v_i / w_i decreasingly.
- Take as much as possible of the item with the most v_i / w_i

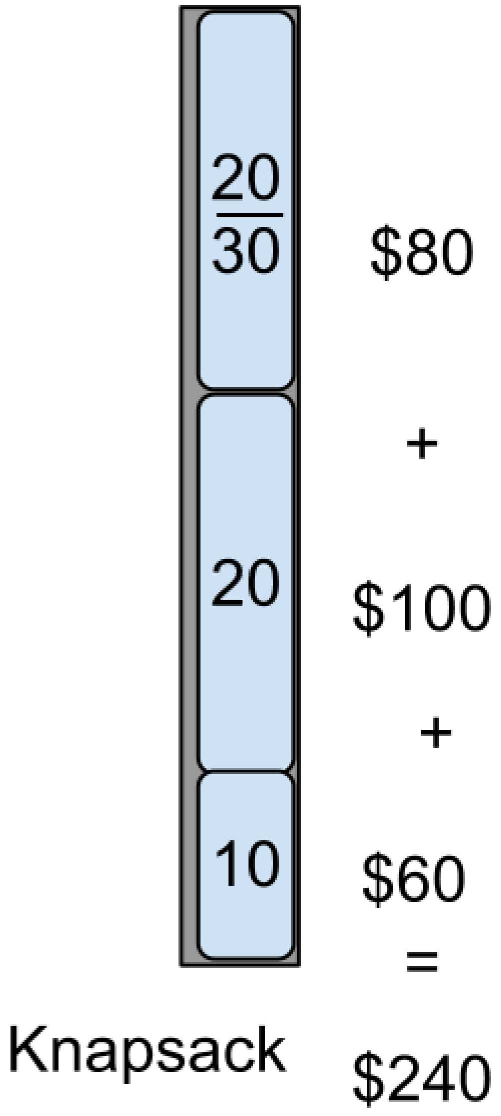
Fractional knapsack(W,S)

```
Sort S, decreasingly according to v_i / w_i;
x[1..n] = 0; //x[i] = amount of i to be taken
weight = 0; i =1;

while (weight < W and i ≤ n)
   if weight + w[i] ≤ W {
     x[i] = 1;
     weight += w[i];
     i++
   } else {
     x[i] = (W - weight) / w[i];
      weight = W;
  }
return x
```



Fractional knapsack(W,S)

```
Sort S, decreasingly according to v_i / w_i;
x[1..n] = 0; //x[i] = amount of i to be taken
weight = 0; i =1;

while (weight < W and i ≤ n)
   if weight + w[i] ≤ W {
     x[i] = 1;
     weight += w[i];
     i++
   } else {
     x[i] = (W - weight) / w[i];
       weight = W;
  }
return x
```



Running time:

```
Fractional knapsack(W,S)
  Sort S, decreasingly
  according to v_i / w_i;
  for(i =1; i ≤ n; i++)
      x[i] =0;
  Weight = 0; i =1;
  while (weight < W and i ≤ n)
     if weight + w[i] ≤ W
       then x[i] = 1;
        {weight = weight + w[i];
         i=i+1;}
       else
         {x[i] = (w - weight) / w[i];
         weight = W;}
     return x
```

O(n log n) — Sort S, decreasingly according to v_i / w_i;

O(n) — for loop

O(n) — while loop

$T(n) = O(n \log n)$.