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

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

- Let's try a recursive approach...

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

- $S(i,s)$:= number of $x \in \{0,1\}^i$ such that $\sum_{j \le i} w_j\ x_j = s$
- Structure of solutions: $S(i,s) = S(i\text{-}1,s) + S(i\text{-}1,s–w_i\ )$ $i = n,...$
- 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 $x \in \{0,1\}^n : \sum w_i\ x_i = t$

- $S(i,s)$:= number of $x \in \{0,1\}^i$ such that $\sum_{j \le i} w_j\ x_j = s$
- Structure of solutions: $S(i,s) = S(i-1,s) + S(i-1,s-w_i)$ $i = n,...$
- Gives recursive approach: $T(n) = 2\ T(n-1) \Rightarrow T(n) \ge 2^n$

- How can we do faster when $k = n$?

- ?

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

- $S(i,s)$:= number of $x \in \{0,1\}^i$ such that $\sum_{j \le i} w_j x_j = s$
- Structure of solutions: $S(i,s) = S(i-1,s) + S(i-1,s-w_i)$ $i = n,...$
- Gives recursive approach: $T(n) = 2\ T(n-1) \Rightarrow T(n) \ge 2^n$

- How can we do faster when $k = n$?

- Stop solving the same problems over and over again!

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

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

- $S(i,s)$:= number of $x \in \{0,1\}^i$ such that $\sum_{j \le i} w_j x_j = s$
- Structure of solutions: $S(i,s) = S(i-1,s) + S(i-1,s-w_i)$ $i = n,...$
- Gives recursive approach: $T(n) = 2\ T(n-1) \Rightarrow T(n) \ge 2^n$

- How can we do faster when $k = n$?

- Stop solving the same problems over and over again!

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

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

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 $0 \le w_i \le k$
- Output: Number of $x \in \{0,1\}^n : \sum w_i x_i = t$

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 $x \in \{0,1\}^n : \sum w_i x_i = t$

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

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

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)
- Space: Trivial: kn^2 Better: ??

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

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)
- Space: Trivial: kn^2 Better: O(kn), just keep two columns

- Steps for dynamic programming approach:

- Identify subproblems (here S(i,s))

- Count subproblems (here kn^2)

- Obtain recursion (here $S(i,s) = S(i-1,s) + S(i-1,s-w_i)$)
  (aka structure of solutions, optimal substructure property)

- The algorithm solves all the subproblems

- Running time = Number of subproblems (here kn^2)
  x Time to compute recursion (here O(1)

- 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

- Subproblems: Cost[c] := number of coins to obtain c

- Number of subproblems: 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: 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: $Cost[c] = 1 + \min_{i \leq k} Cost[c - d_i]$

Algorithm:

- Initialize vector Cost to ∞
- 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: $Cost[c] = 1 + \min_{i \le k} Cost[c - d_i]$

Algorithm:

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

- T(n) = kn

- To know which coins to use, store argmin
  Can reconstruct solution backwards

- Dynamic programming in economics

- Let us plan Bob's next L years.

- At the beginning of each year he owns savings + wage

- He must decide how much to consume, rest is saved
  Savings earn interest (1+ρ)
  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

- He starts with savings 0.

Life can only be understood backwards;

but it must be lived forwards.

Soren Kierkegaard

- Formulate as problem

- U[k,i] := utility for years i, i+1, …, L if at beginning of year i
  savings + wage = k

- U[k,L] := ?

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

- Formulate as problem

- U[k,i] := utility for years i, i+1, …, L if at beginning of year i
  savings + wage = k

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

- U[k,i] := ?

- Formulate as problem

- U[k,i] := utility for years i, i+1, …, L if at beginning of year i
  savings + wage = k

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

- U[k,i] := $\max_c$ log (c) + U[(k - c)(1+ρ) + w, i+1]
  $0 \le c \le k \le wL\,(1+\rho)^L$
  Consumption = argmax

- A recursive algorithm for U[0,0] would take time $T \ge 2^L$
- Dynamic programming takes time poly(L,W,$(1+\rho)^L$)

- 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 savings + wage = k
- U[k,L] := log(k)
- U[k,i] := $\max_c$ log (c) + ?

- 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 savings + wage = k
- U[k,L] := log(k)
- U[k,i] := $\max_c$ log (c) +

  q U[(k - c)(1+ρ) + w, i+1] + (1-q) U[(k - c)(1-ρ) + w, i+1]

- Longest common subsequence

- Given two strings X and 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 $X = (x_1, x_2, \ldots, x_n)$,

  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

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

•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

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

What arrows?

Completed LCS Table

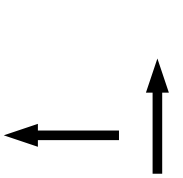
	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	←↑ Ø	↖(G)	←(G)	←(G)	←(G)
A	Ø	↖ (A)	←↑(A) & (G)	←↑(A) & (G)	↖(GA)	←(GA)
C	Ø	↑ (A)	←↑(A) & (G)	↖(AC) & (GC)	←↑(AC) & (GC) & (GA)	←↑(AC) & (GC) & (GA)

The final result is that the last cell contains all the longest

- 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

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



Completed LCS Table

	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)

The final result is that the last cell contains all the longest

- Then we can reconstruct the sequence backwards.

- We have described dynamic programming in an iterative "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 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 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.

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

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

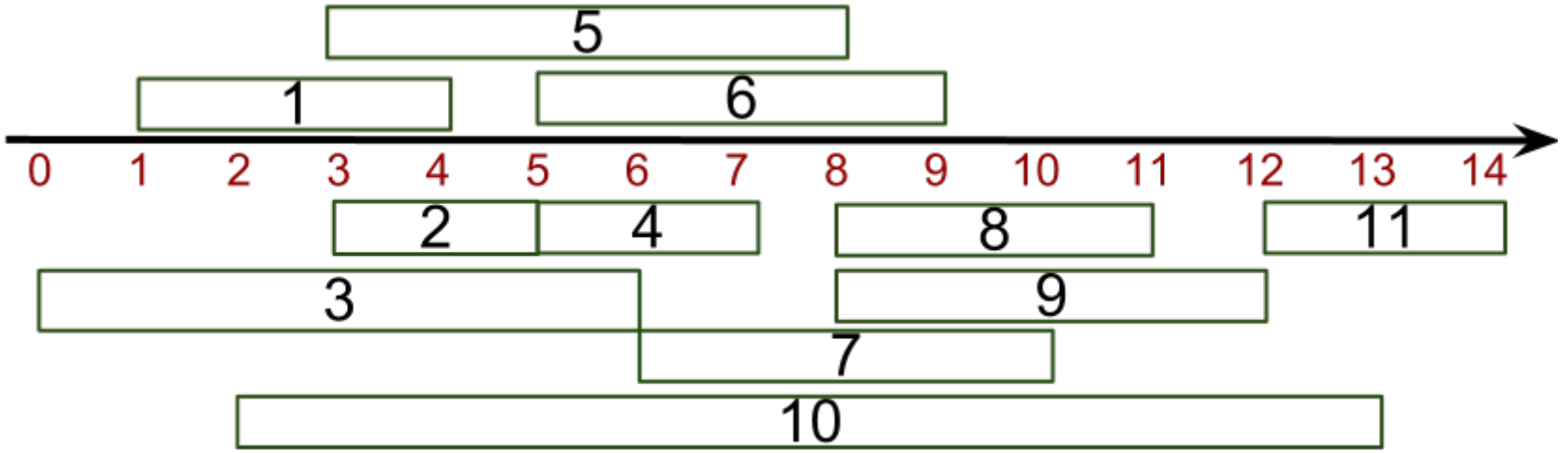
Output:

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

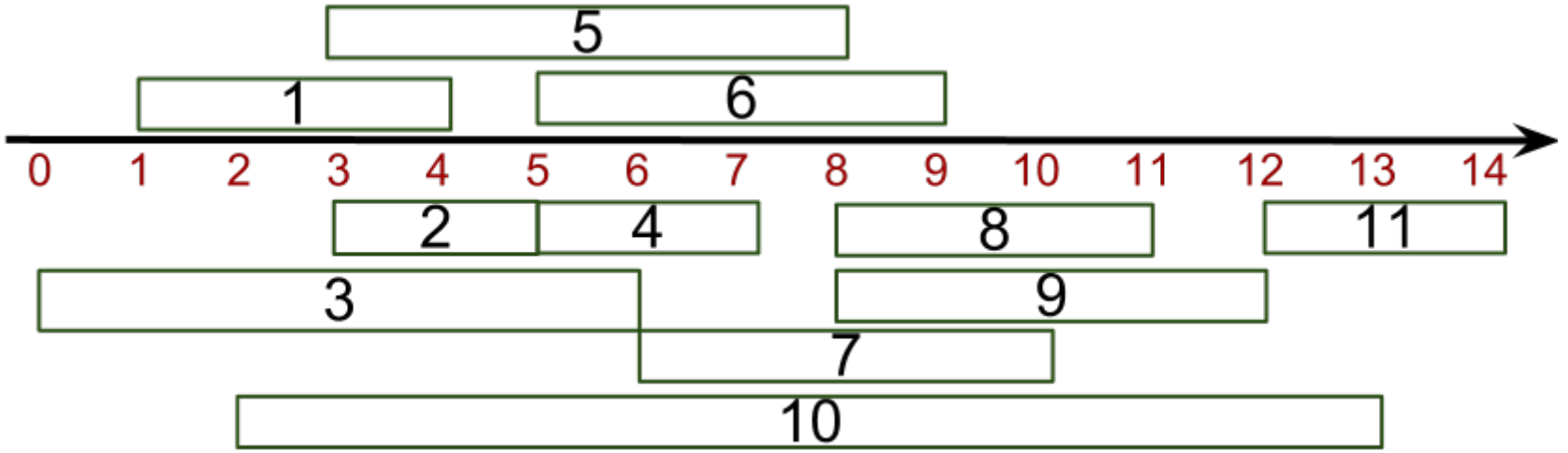
A set of compatible activities = ?



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}).

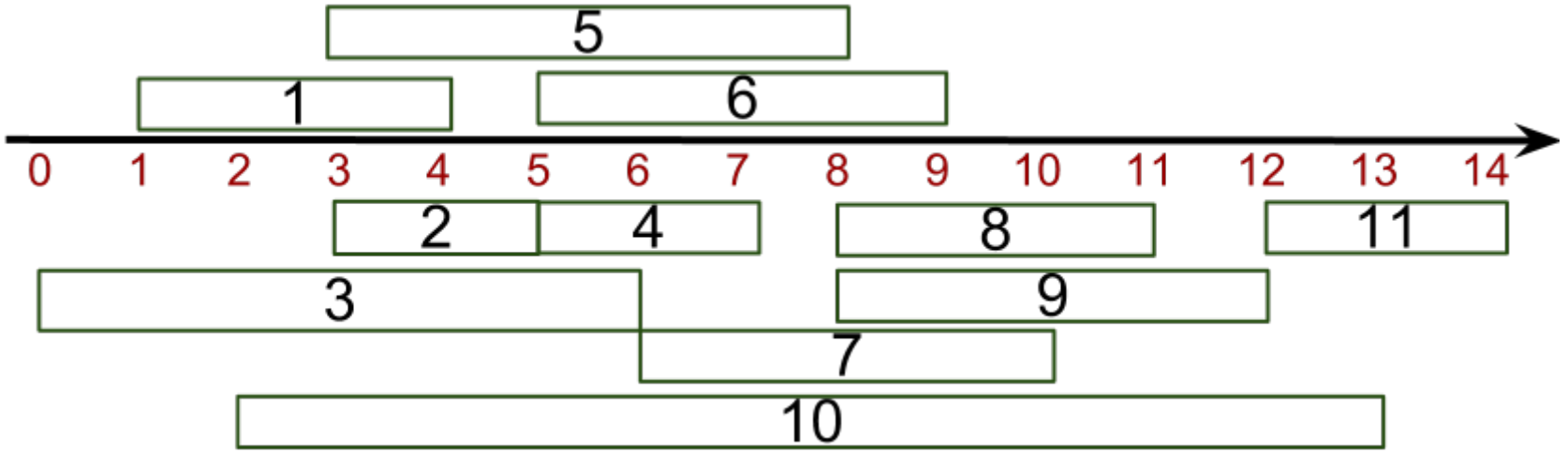


Example:

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 = ?

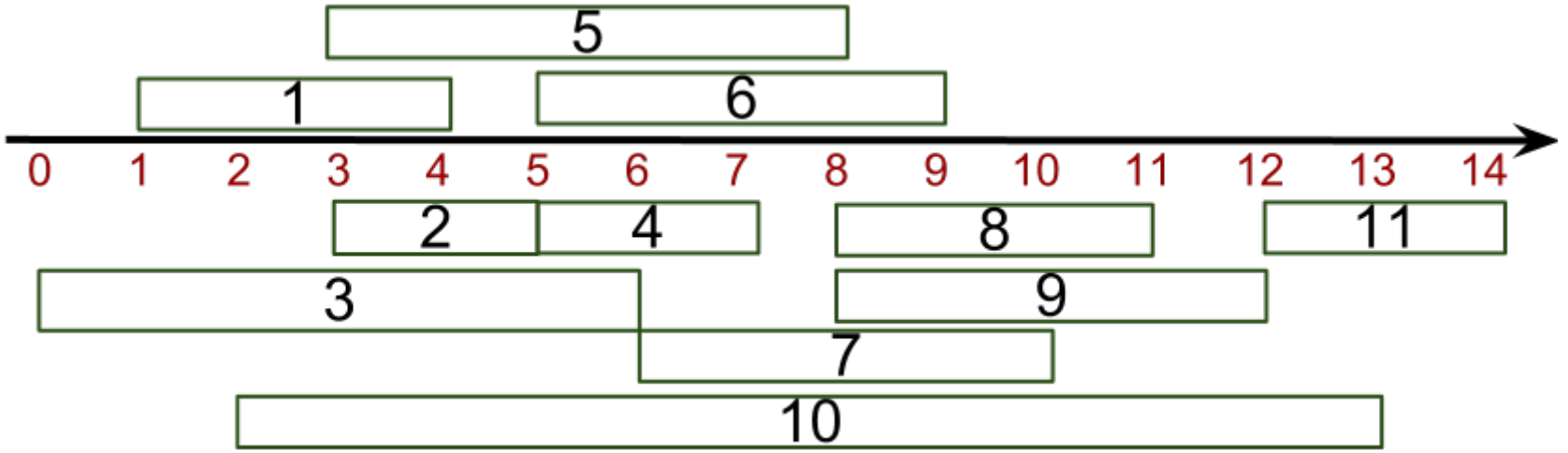


Example:

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}).



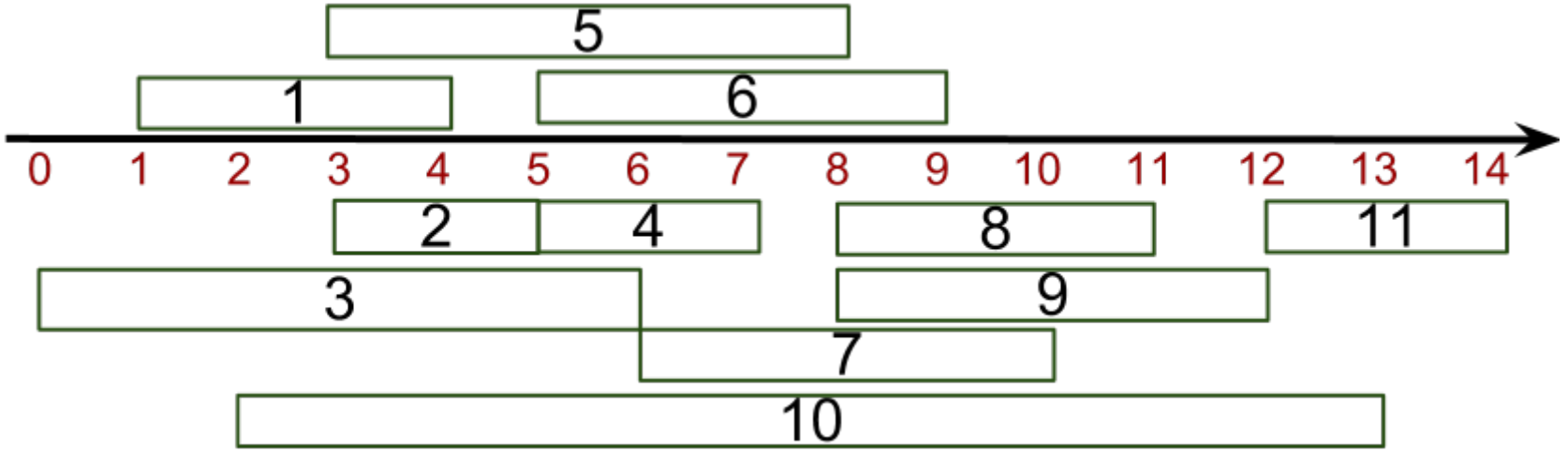
Example:

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 ?



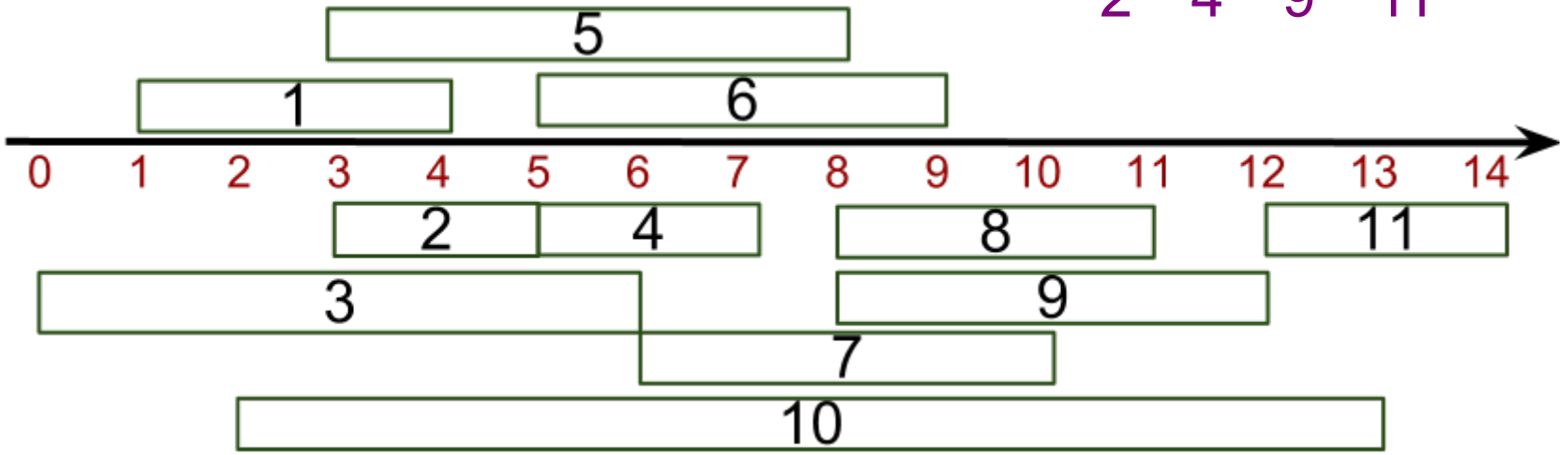
Example:

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

- 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[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

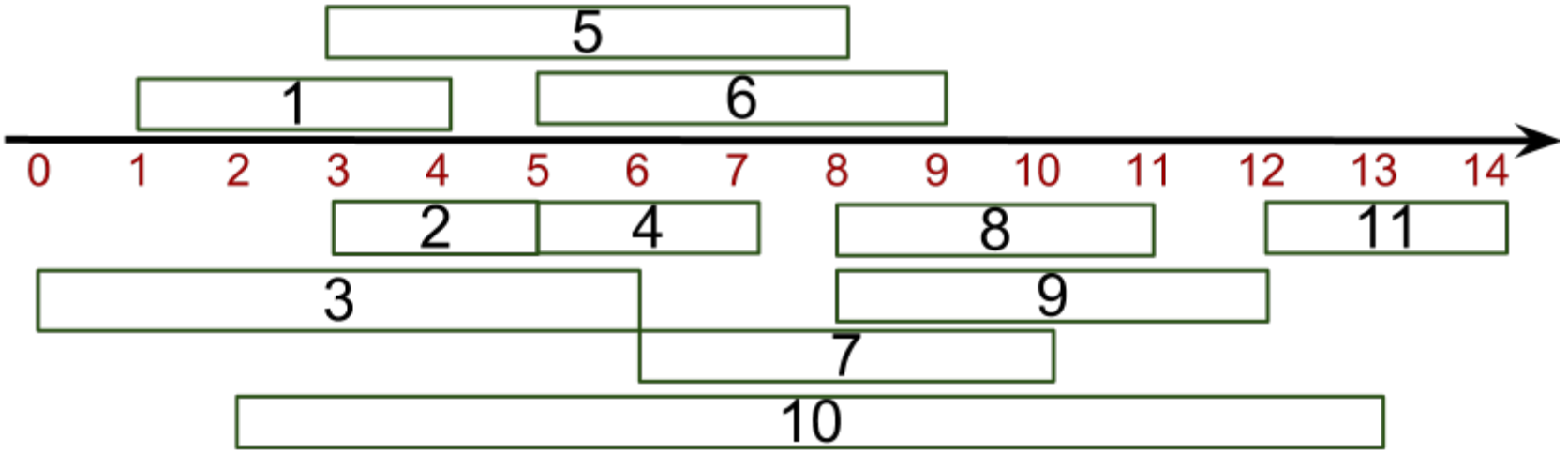
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

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

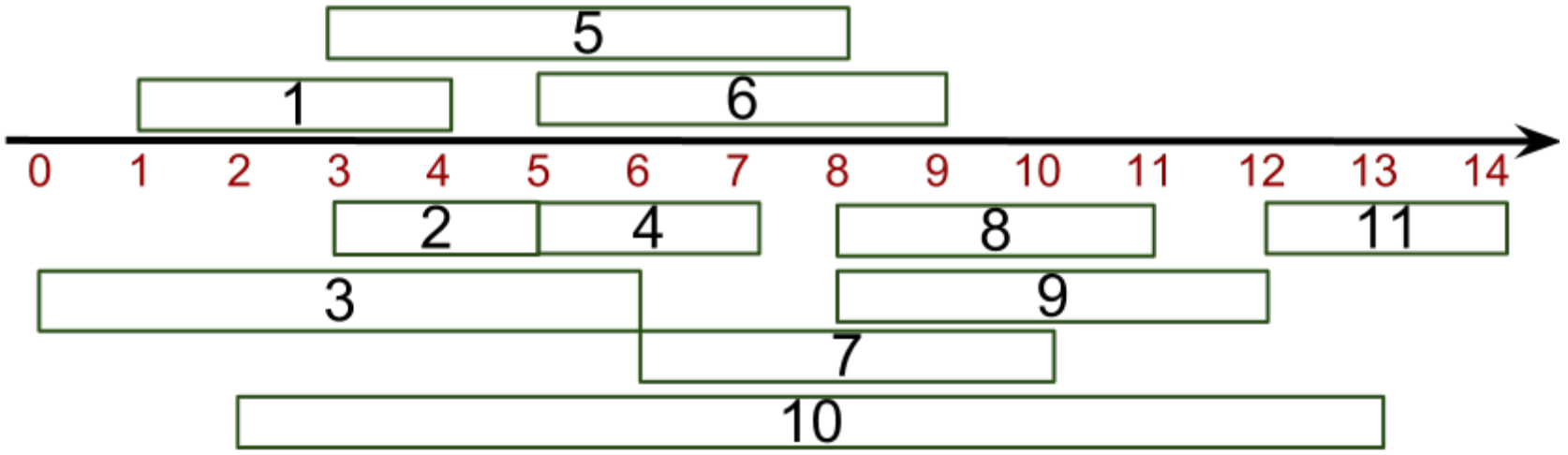


```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

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



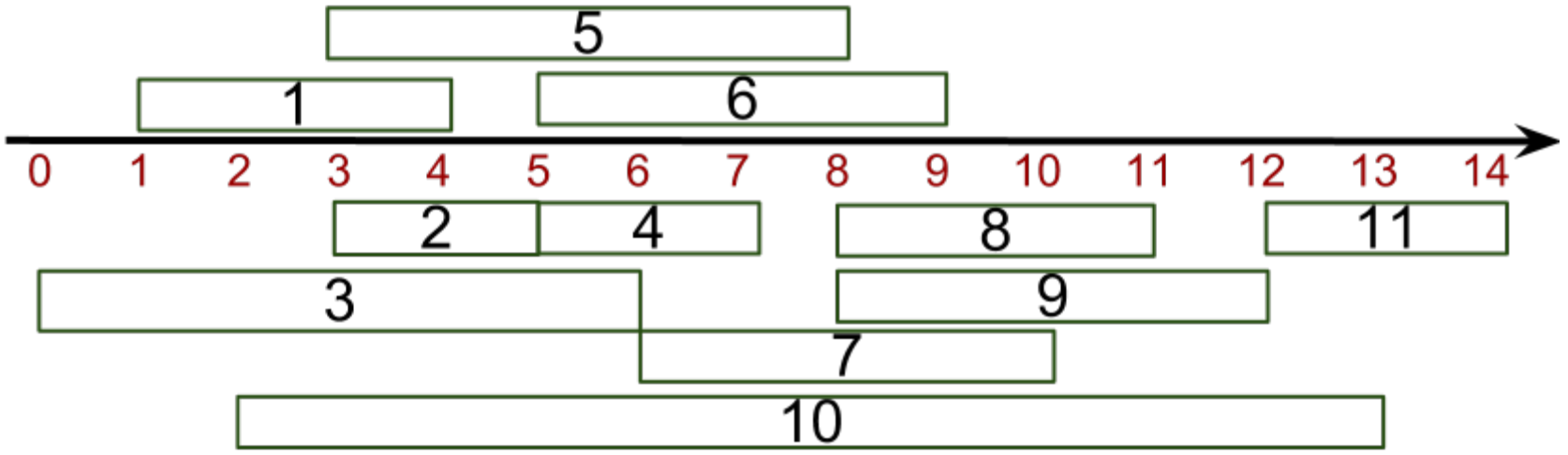
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

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;
}
```

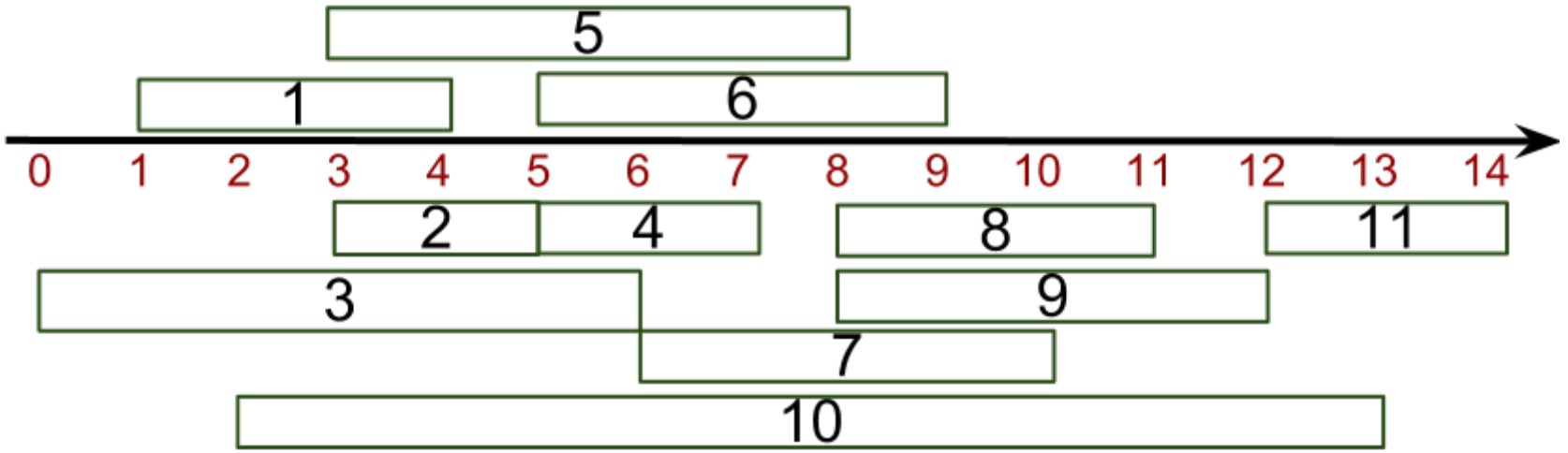
Example:

$S:=\{a_1\}$

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



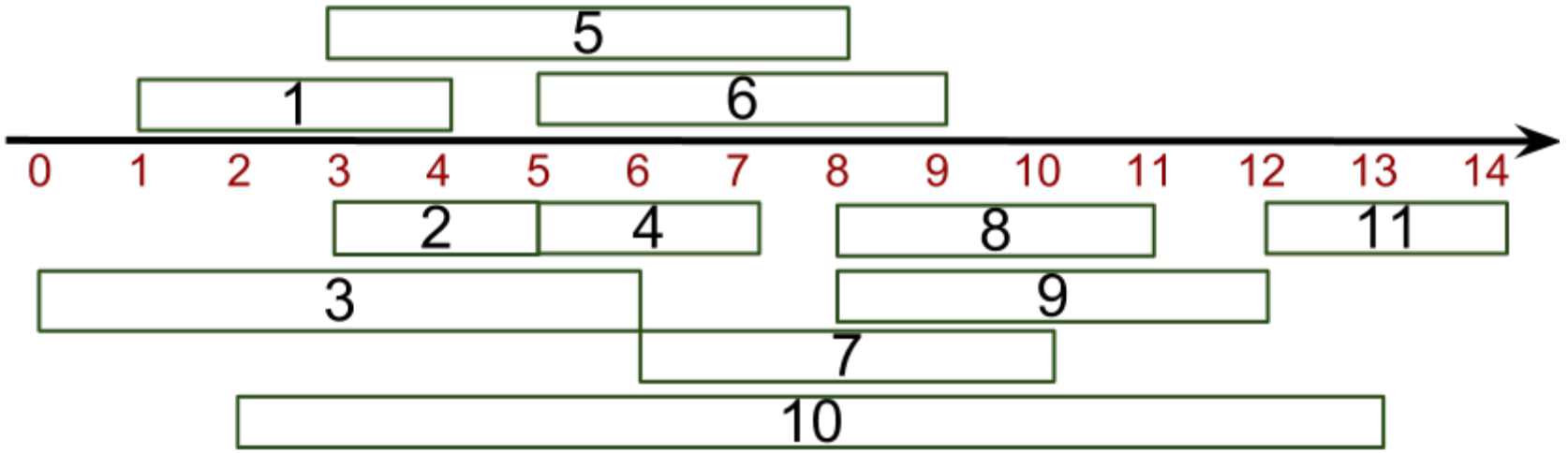
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

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;
}
```

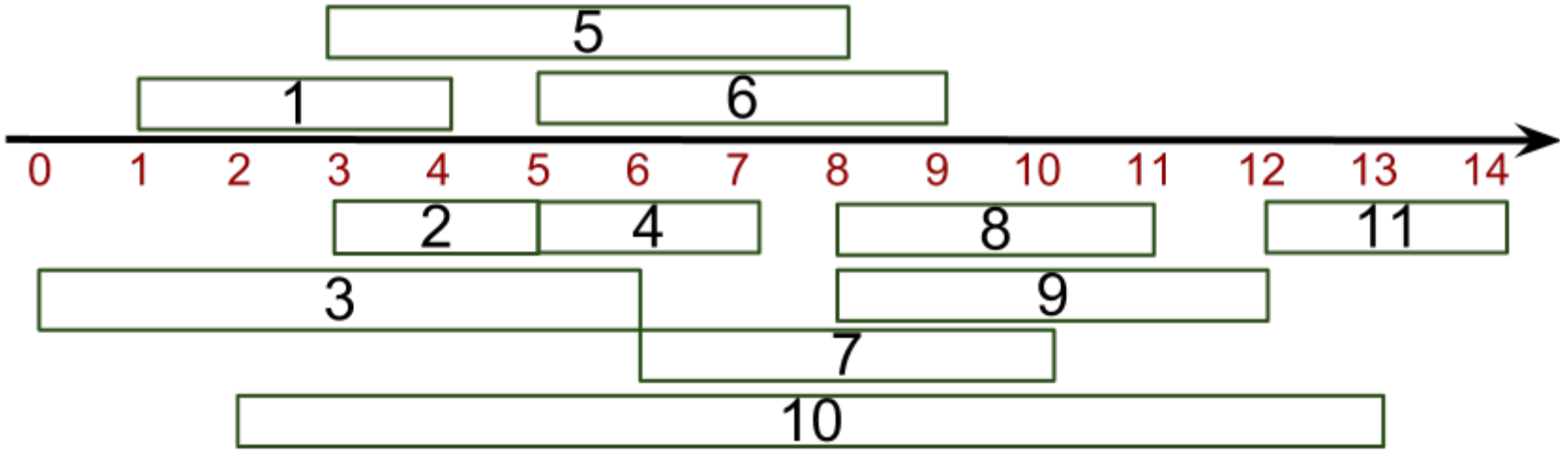
Example:

$S:=\{a_1\}$

s[2] ≥ 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;
}
```

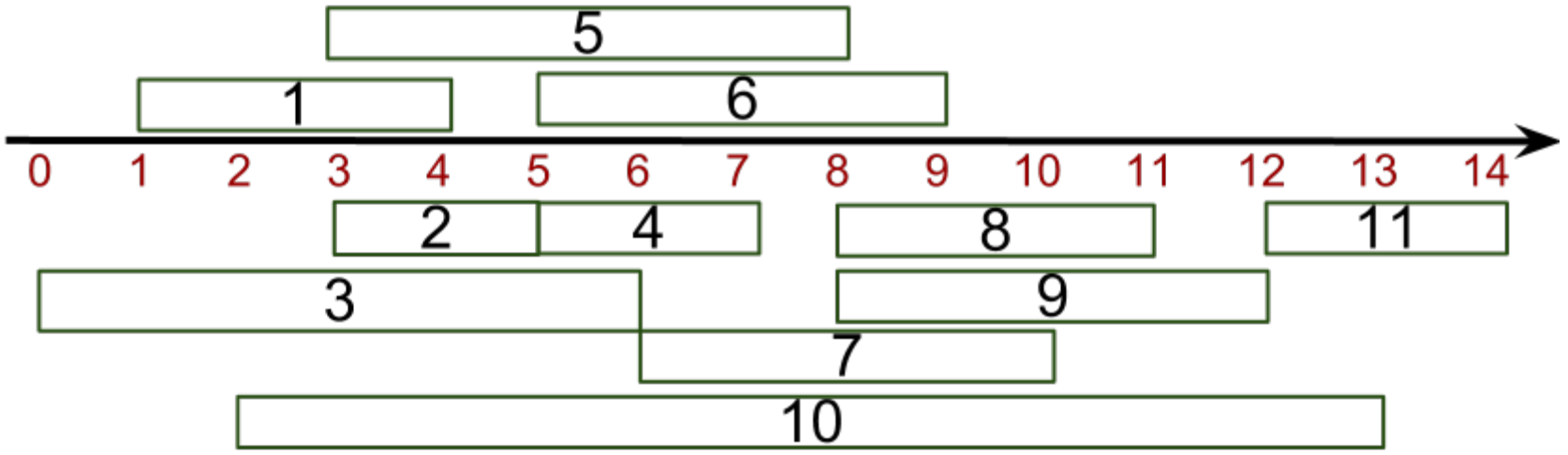
Example:

$S:=\{a_1\}$

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;
}
```

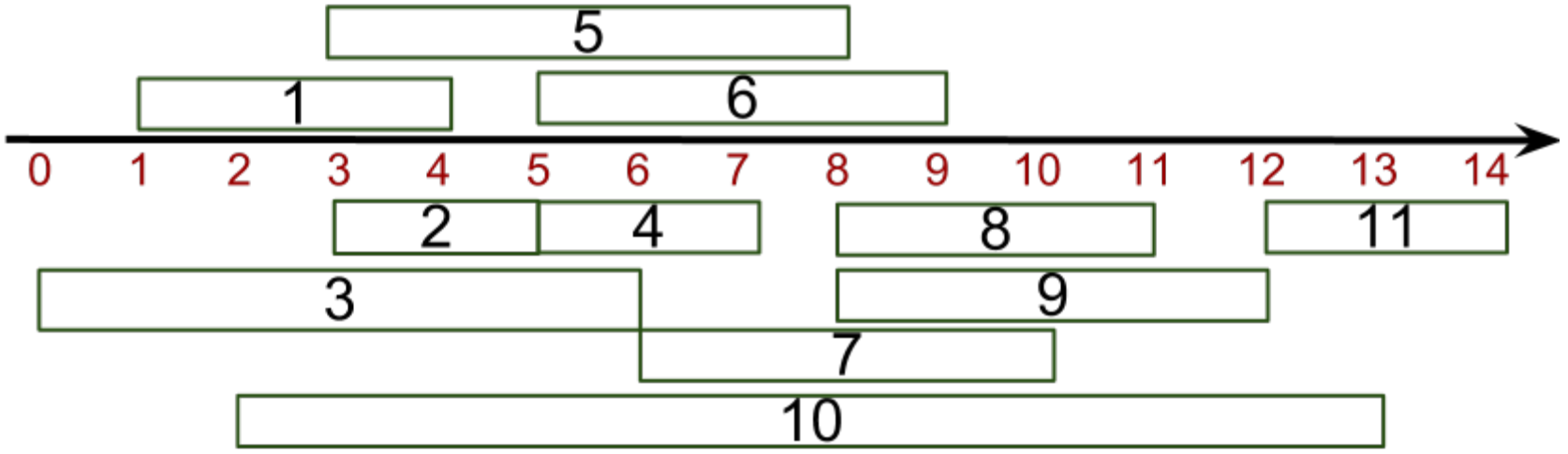
Example:

$S:=\{a_1\}$

s[3] ≥ 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;
}
```

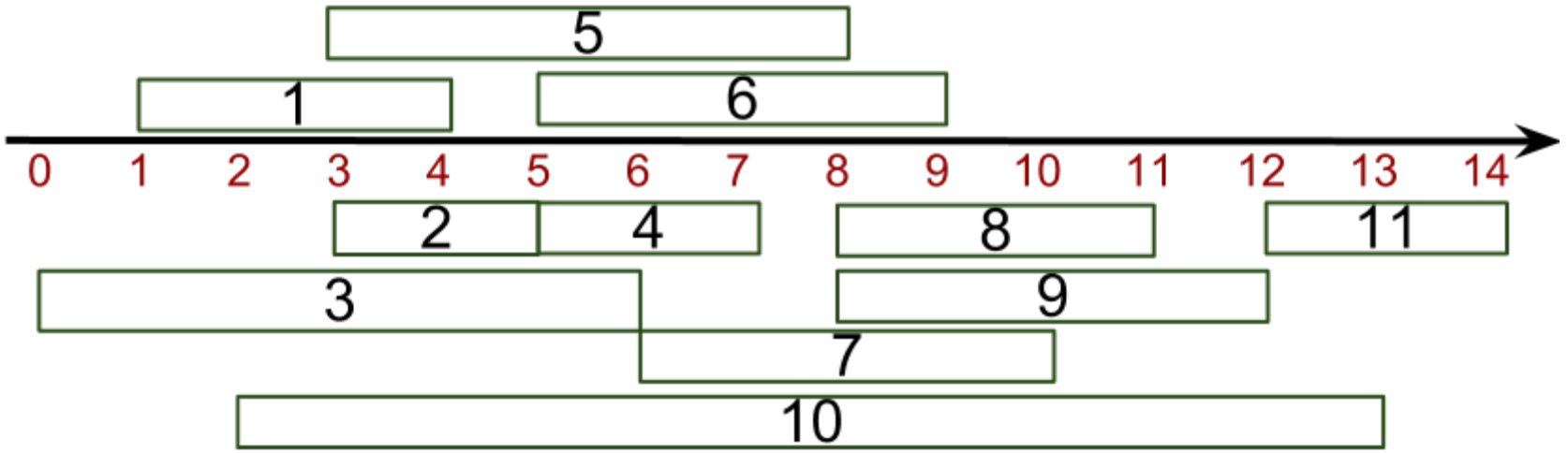
Example:

$S:=\{a_1\}$

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;
}
```

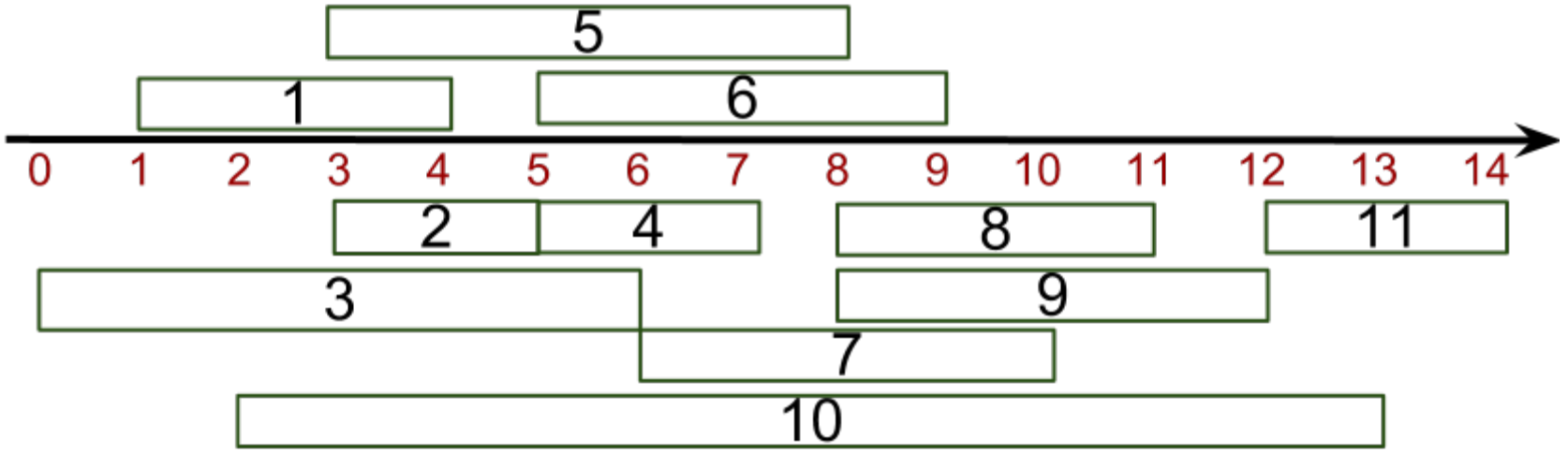
Example:

$S:=\{a_1\}$

s[4] ≥ 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;
}
```

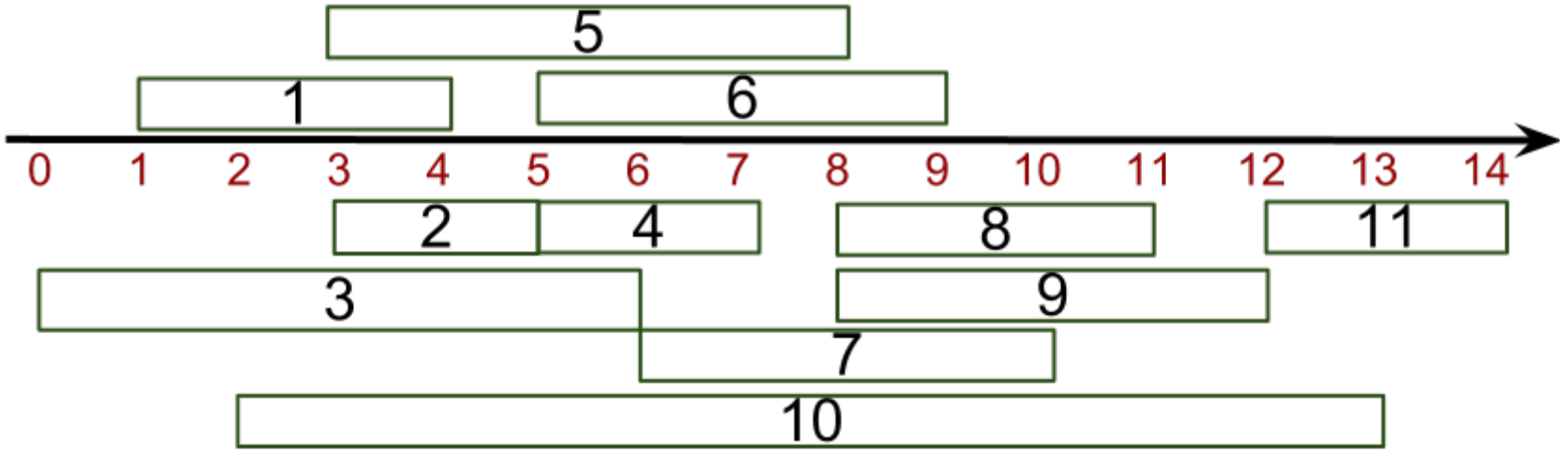
Example:

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

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:

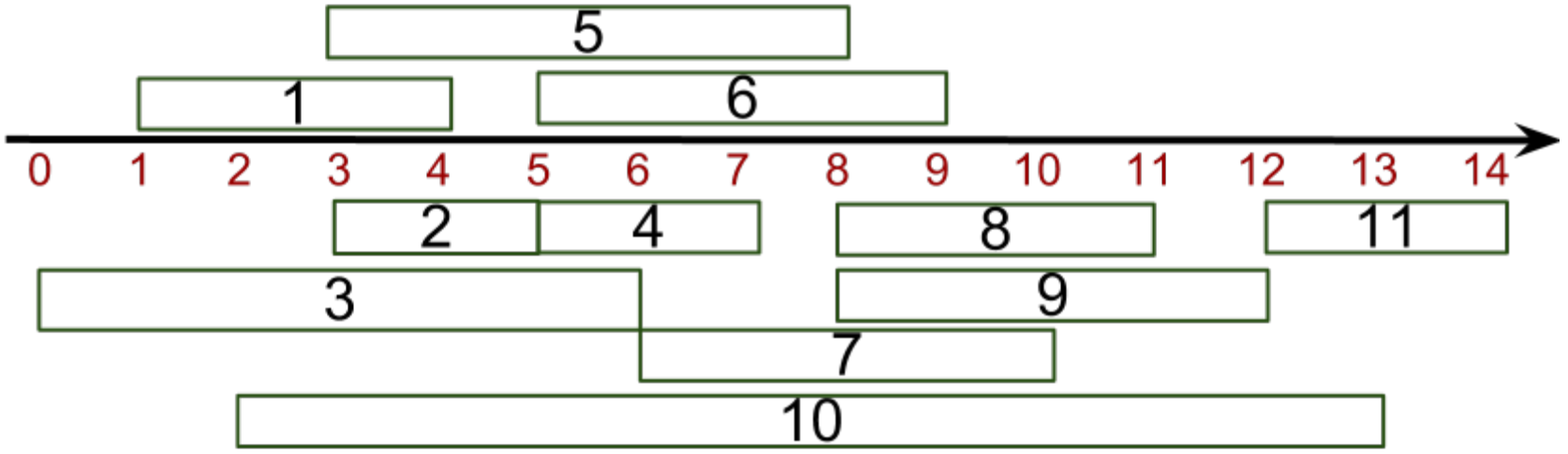
$S:=\{a_1, a_4\}$

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



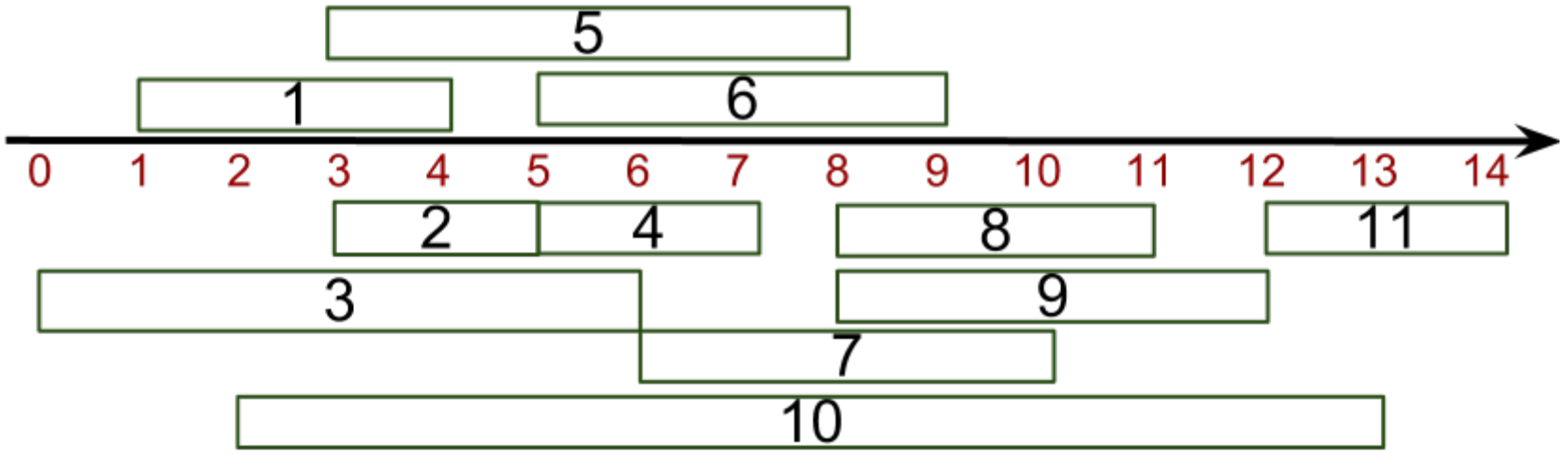
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

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;
}
```

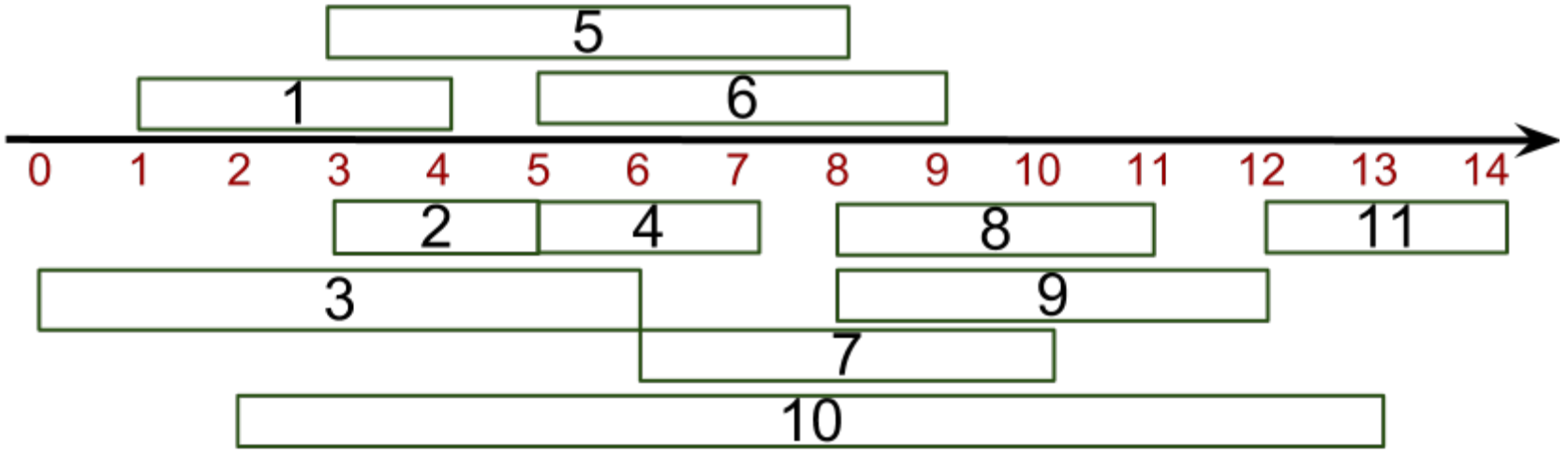
Example:

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

s[5] ≥ 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;
}
```

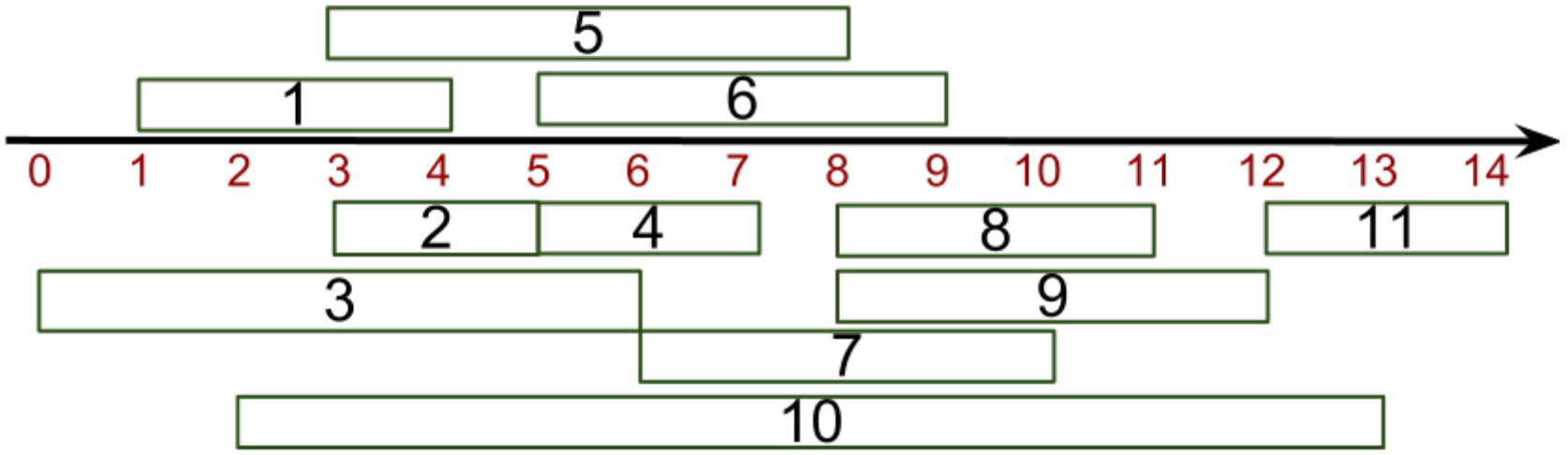
Example:

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

s[5] < f[4]

i 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;
}
```

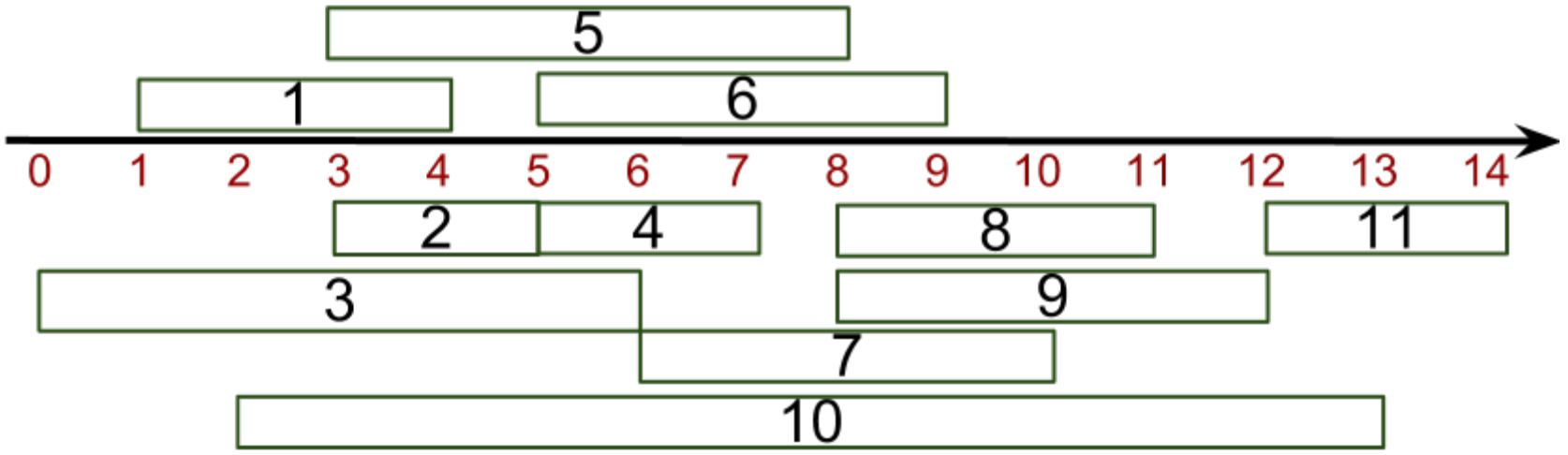
Example:

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

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;
}
```

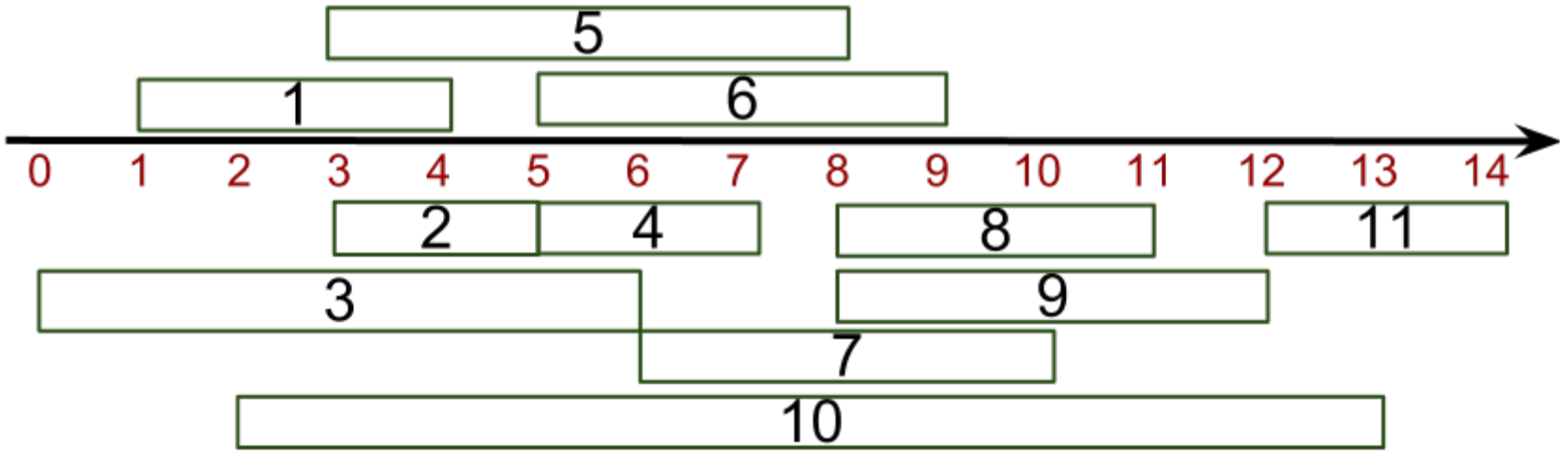
Example:

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

s[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;
}
```

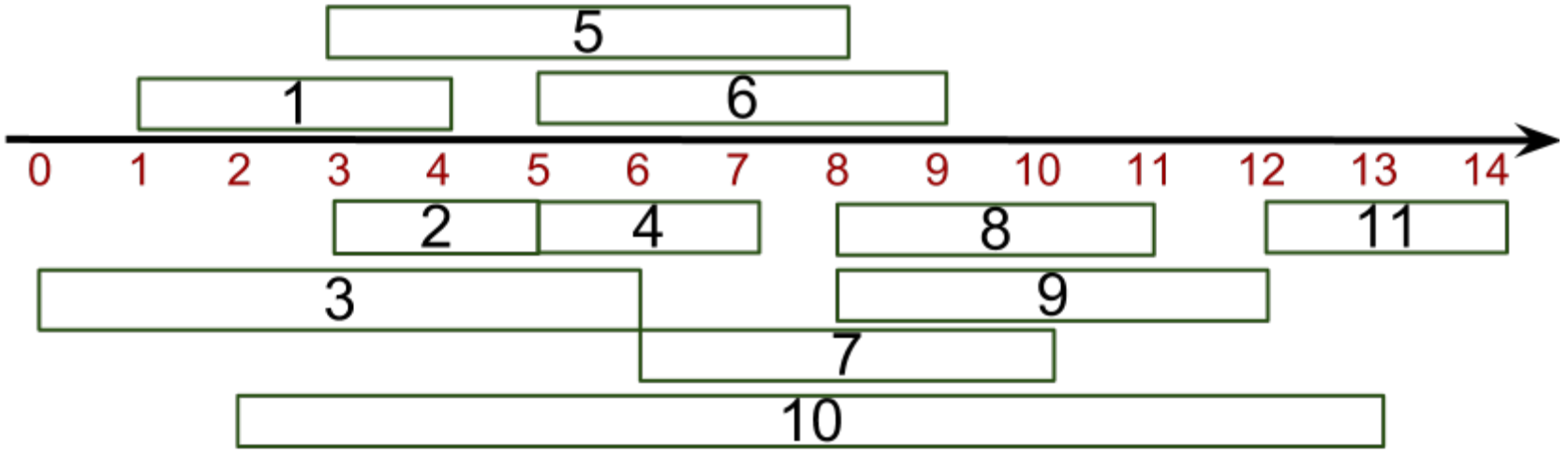
Example:

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

s[7] ≥ 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;
}
```

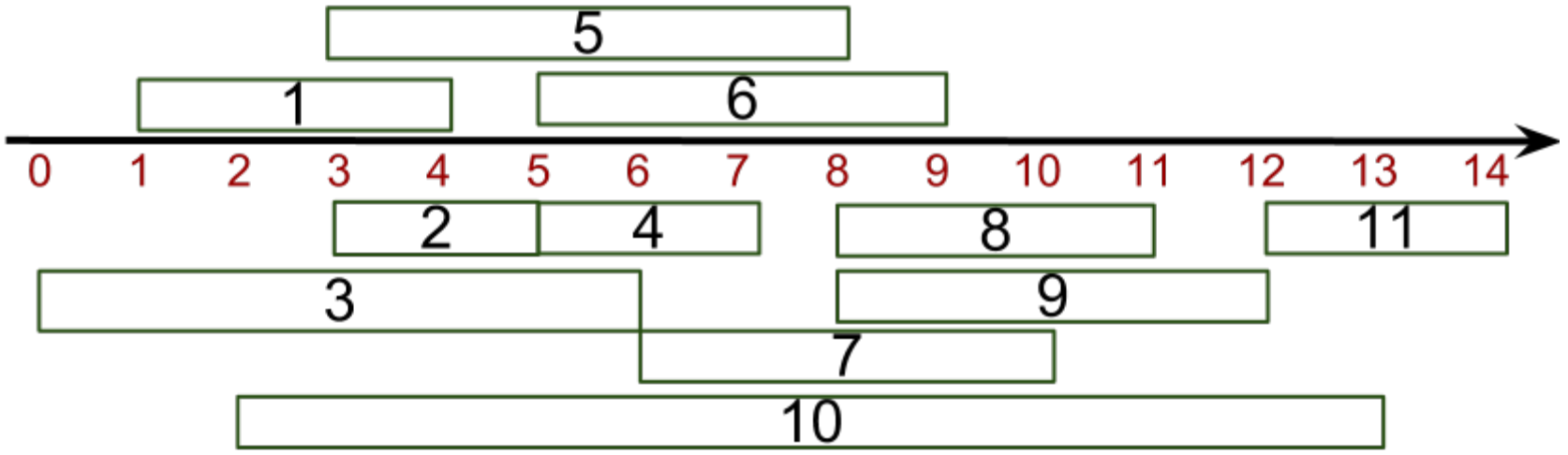
Example:

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

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;
}
```

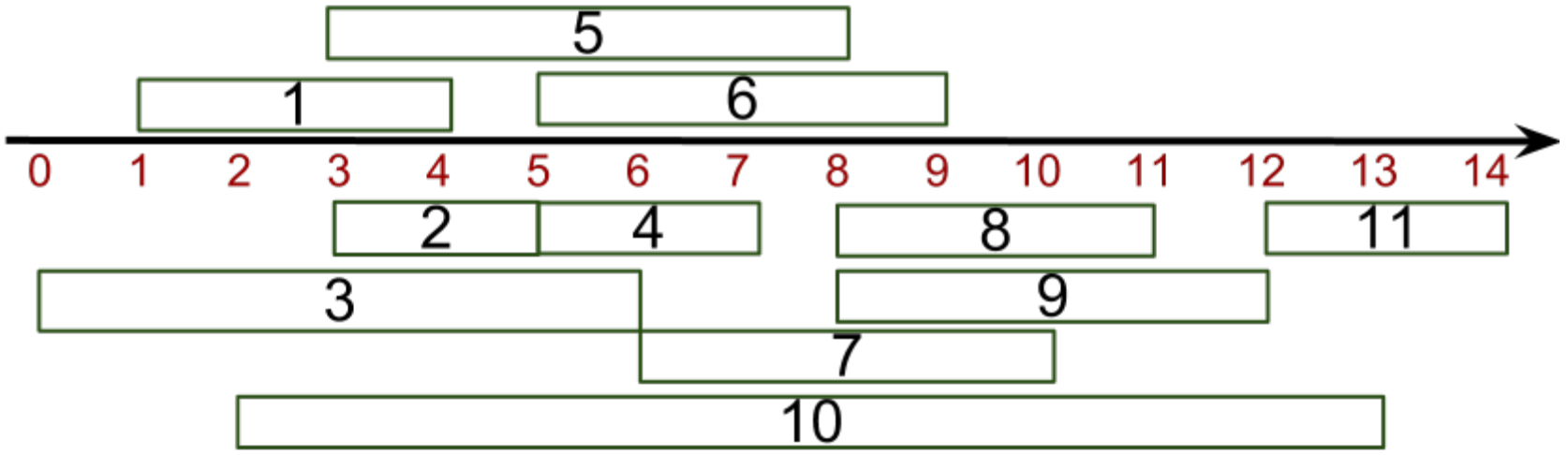
Example:

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

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;
}
```

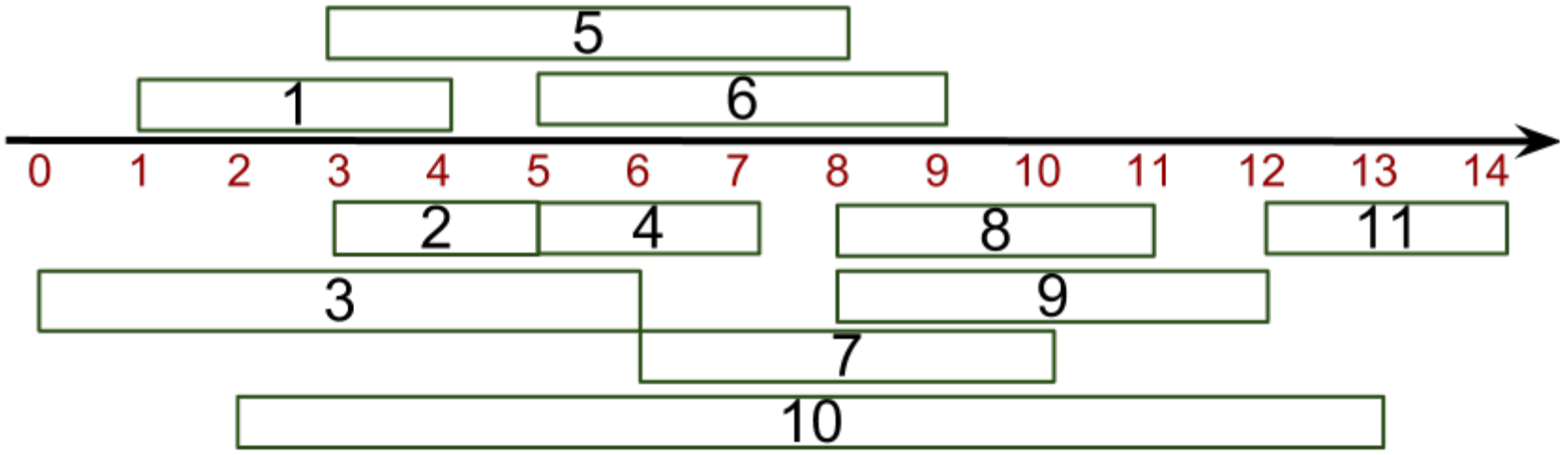
Example:

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

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;
}
```

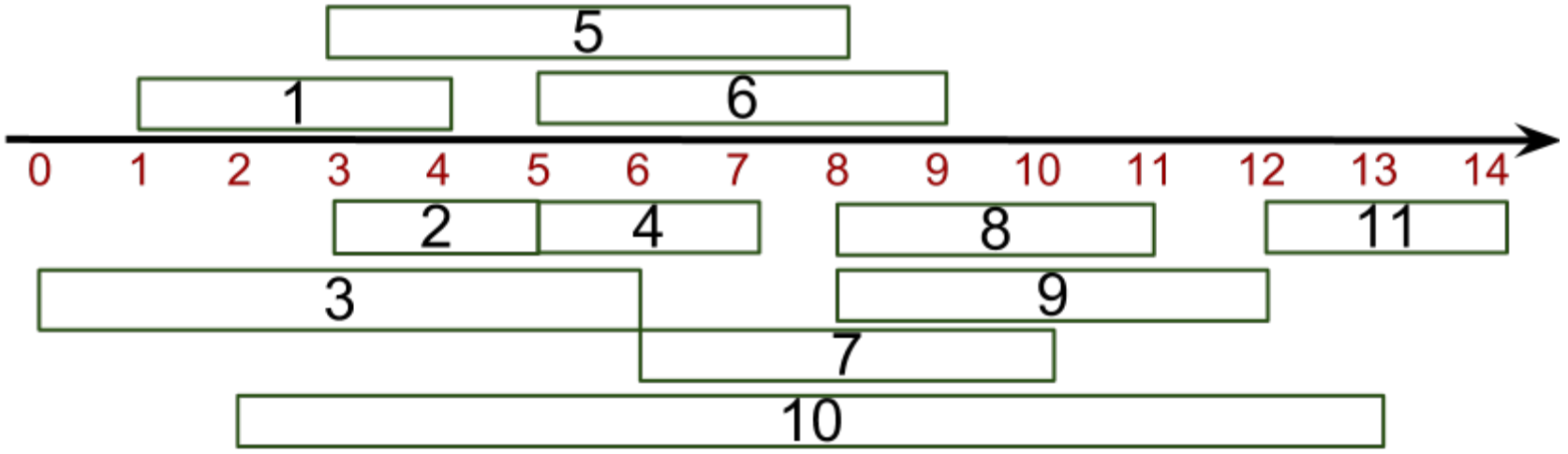
Example:

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

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



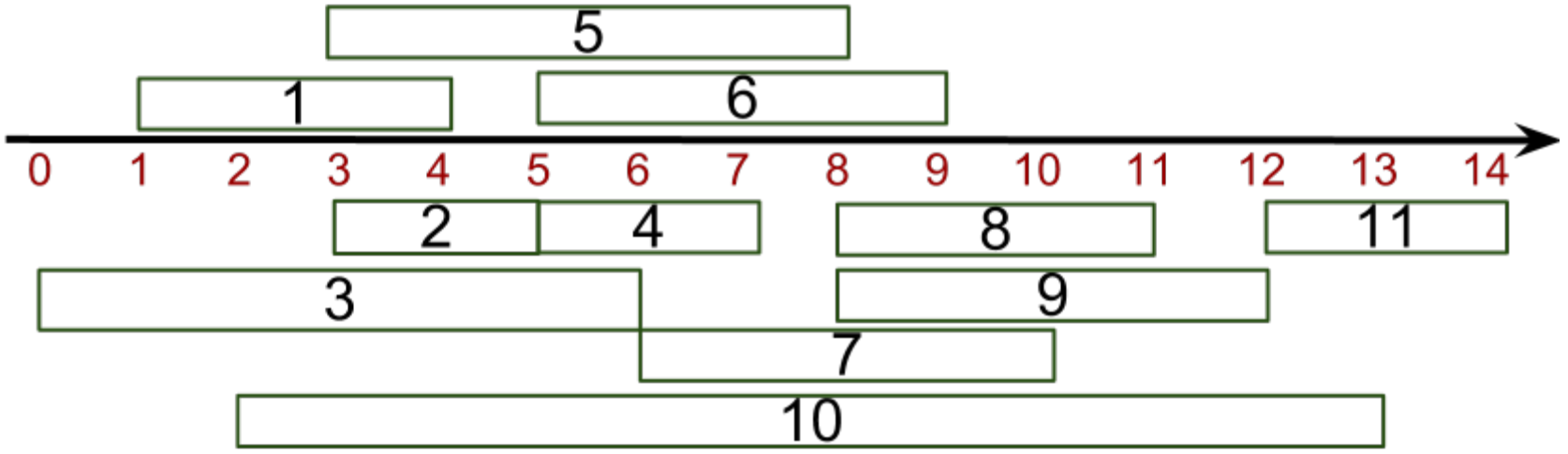
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

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;
}
```

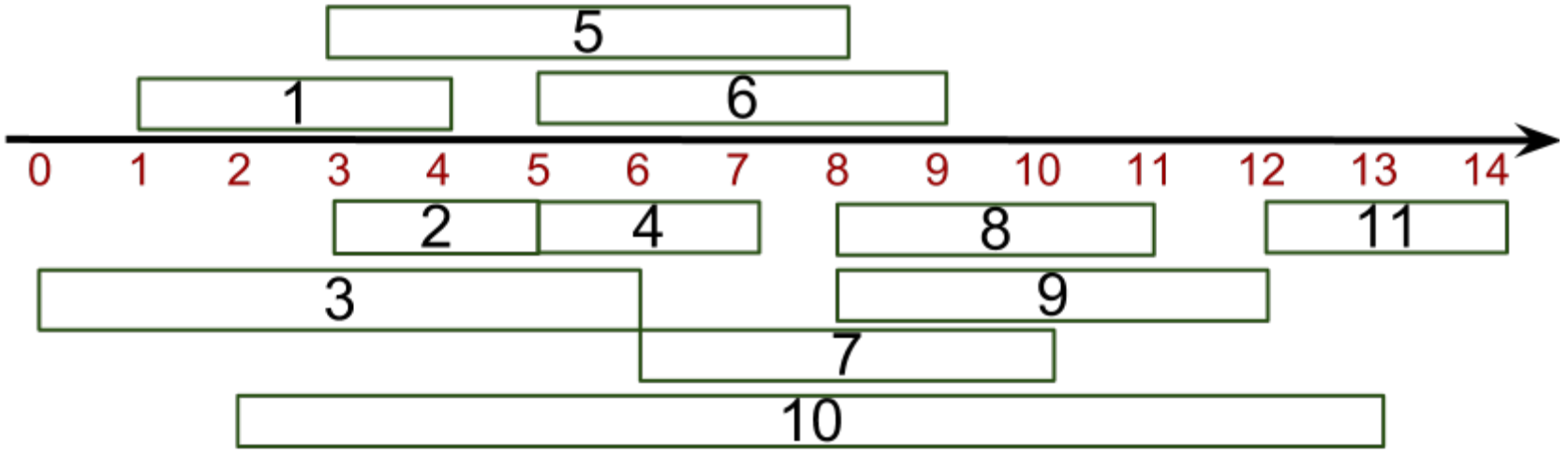
Example:

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

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

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



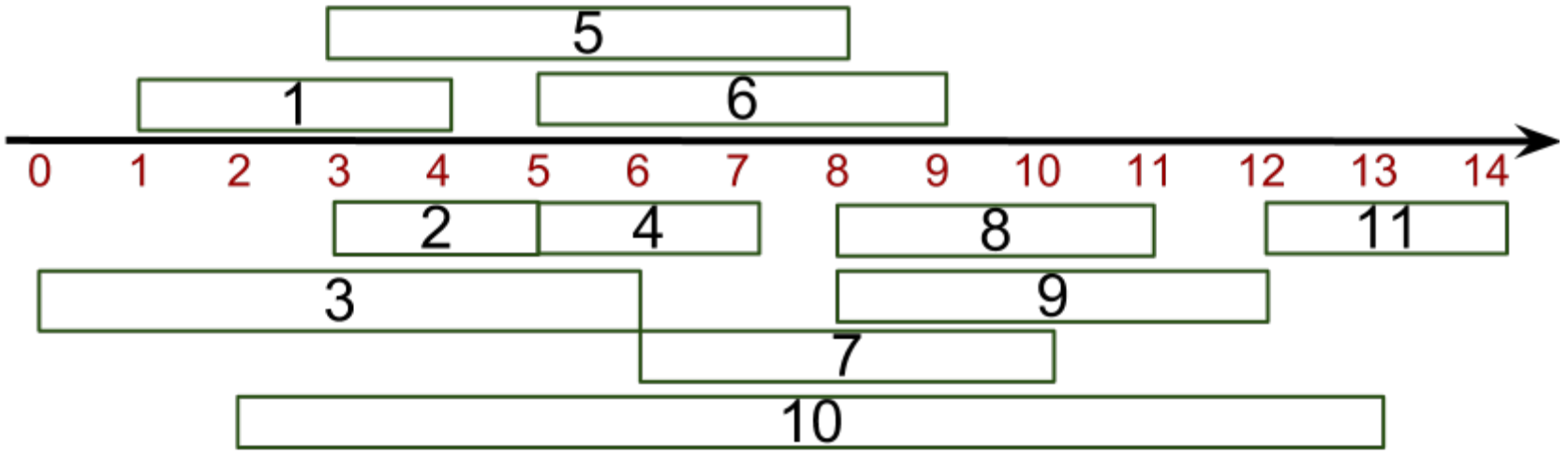
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

Example:

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

$s[9] < f[8]$

								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



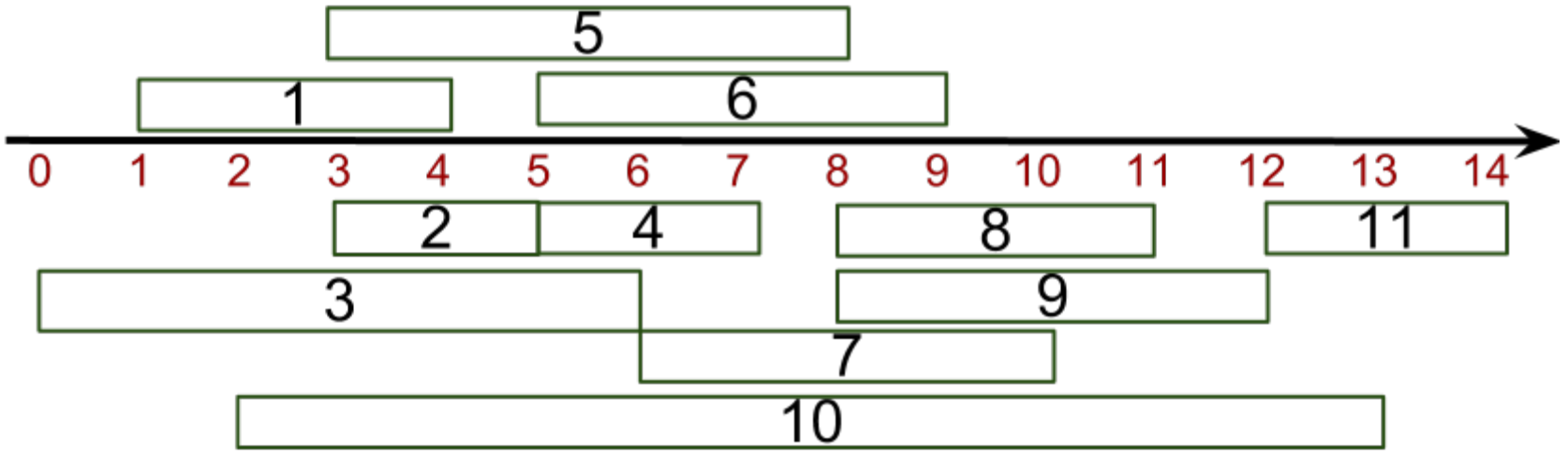
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

Example:

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

s[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;
}
```

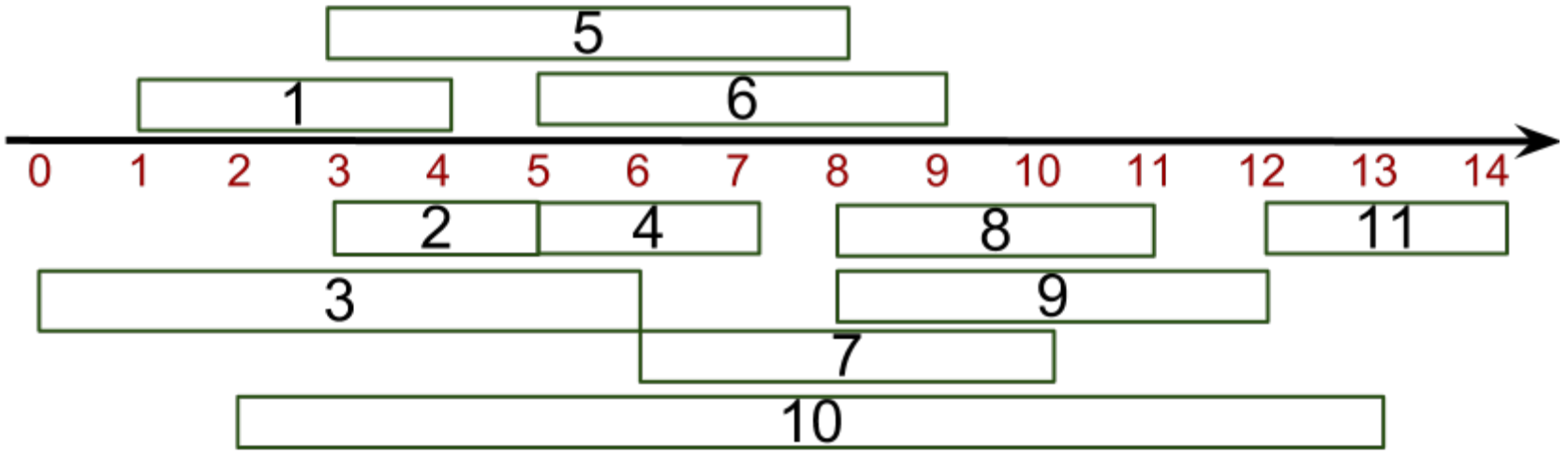
Example:

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

s[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



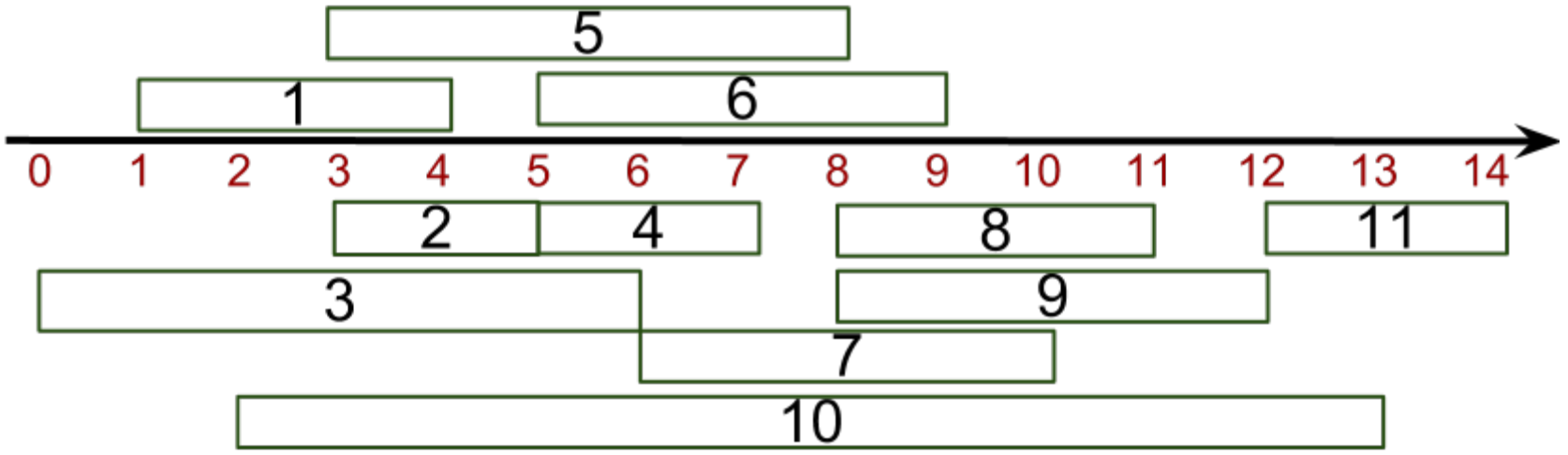
```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

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;
}
```

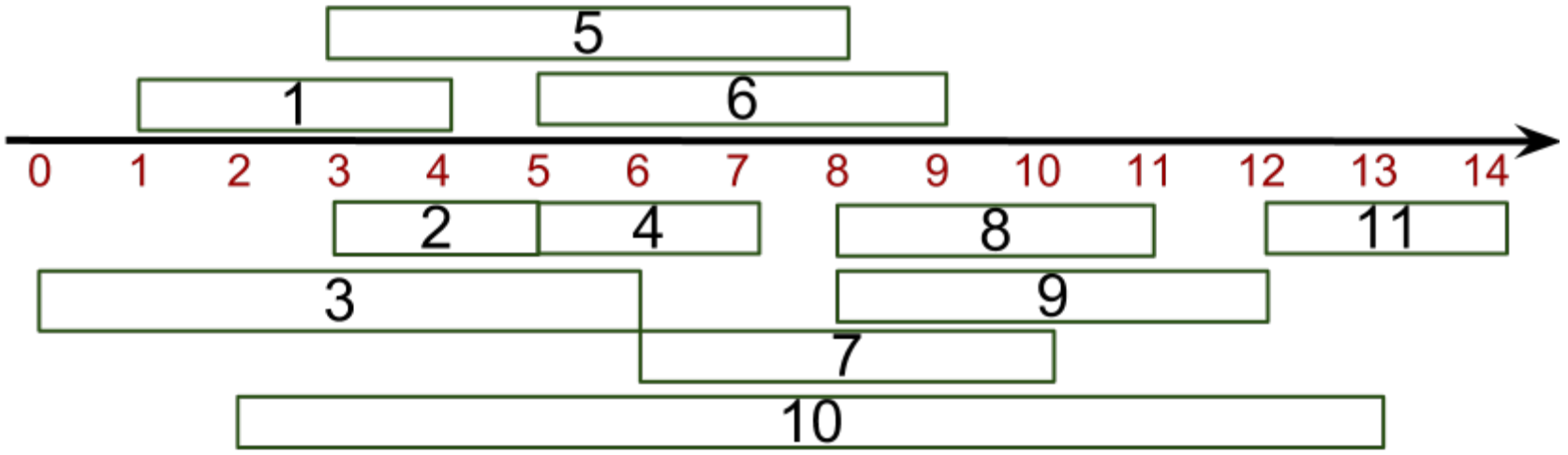
Example:

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

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

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;
}
```

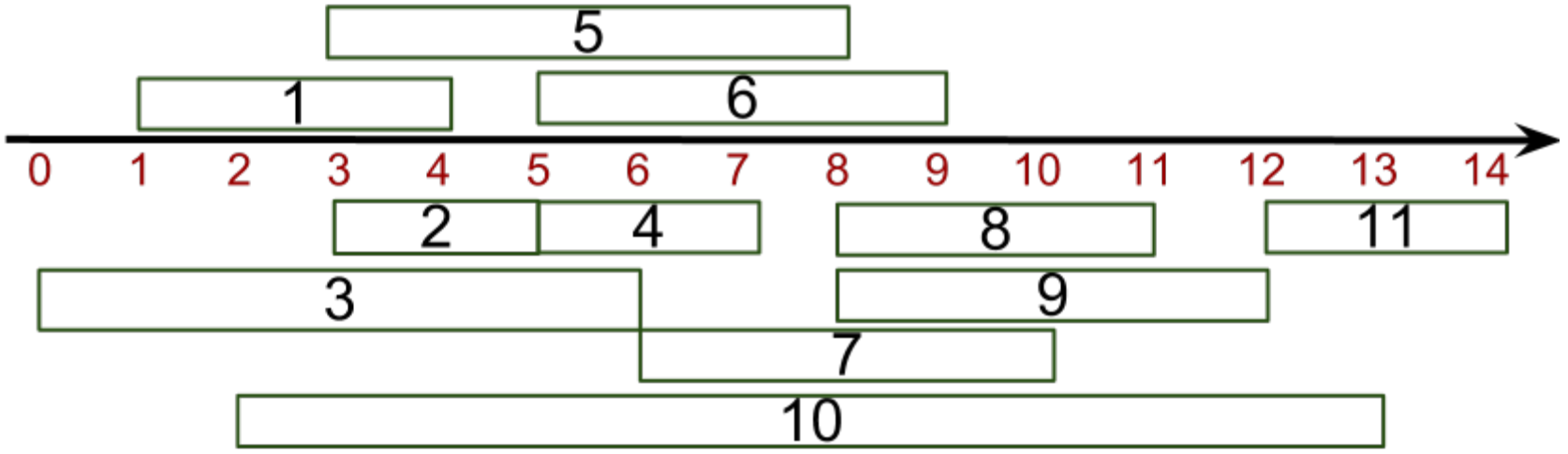
Example:

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

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

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:

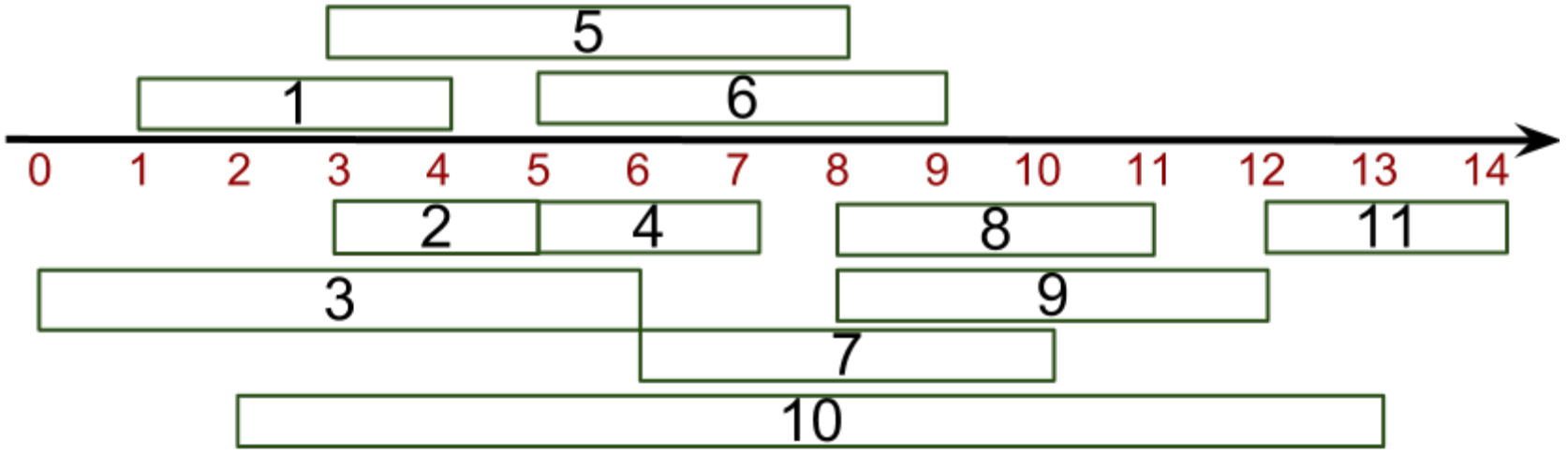
$S:=\{a_1, a_4, a_8, a_{11}\}$

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:

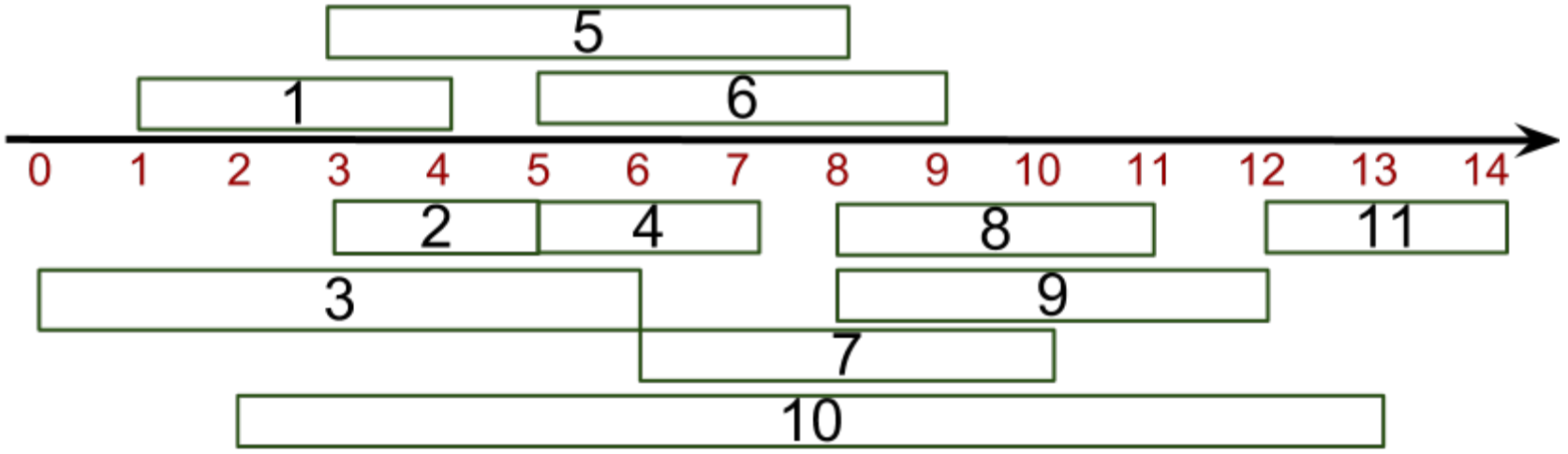
$S:=\{a_1, a_4, a_8, a_{11}\}$

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

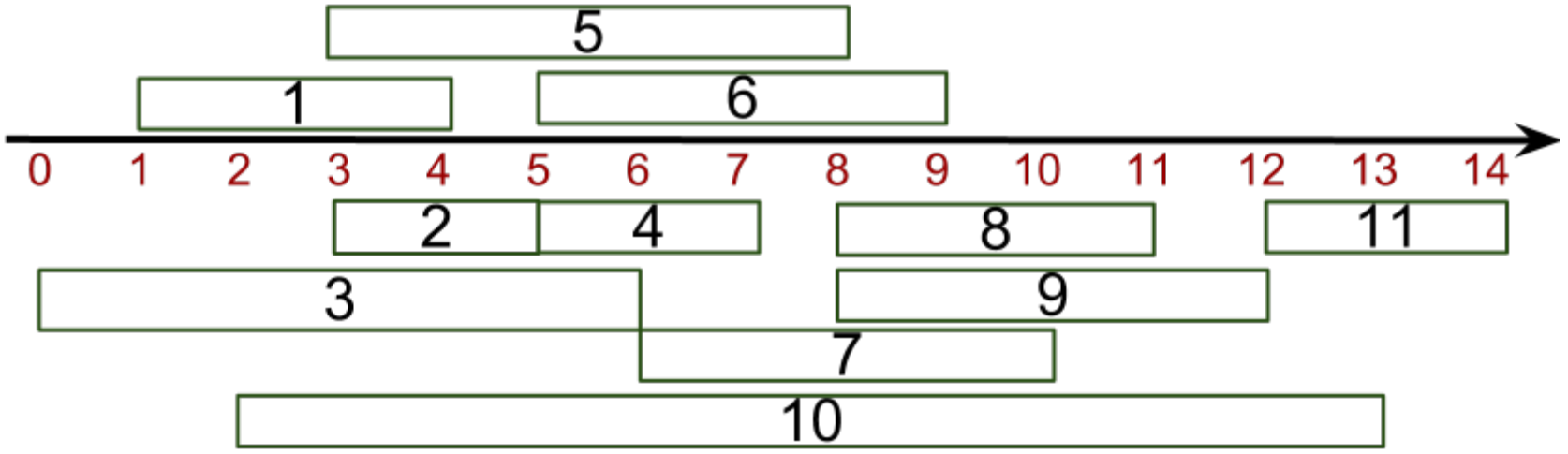


```
activity-selection(A) {
sort A increasingly
according to f [i];
n:= length[A];
S:={a[i]};
i:=1;
for (m=2; m ≤ n; m++)
     if (s[m] ≥ f[i] ) {
          Add a[i] to S;
          i :=m;} return S;
}
```

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



Knapsack

Input:

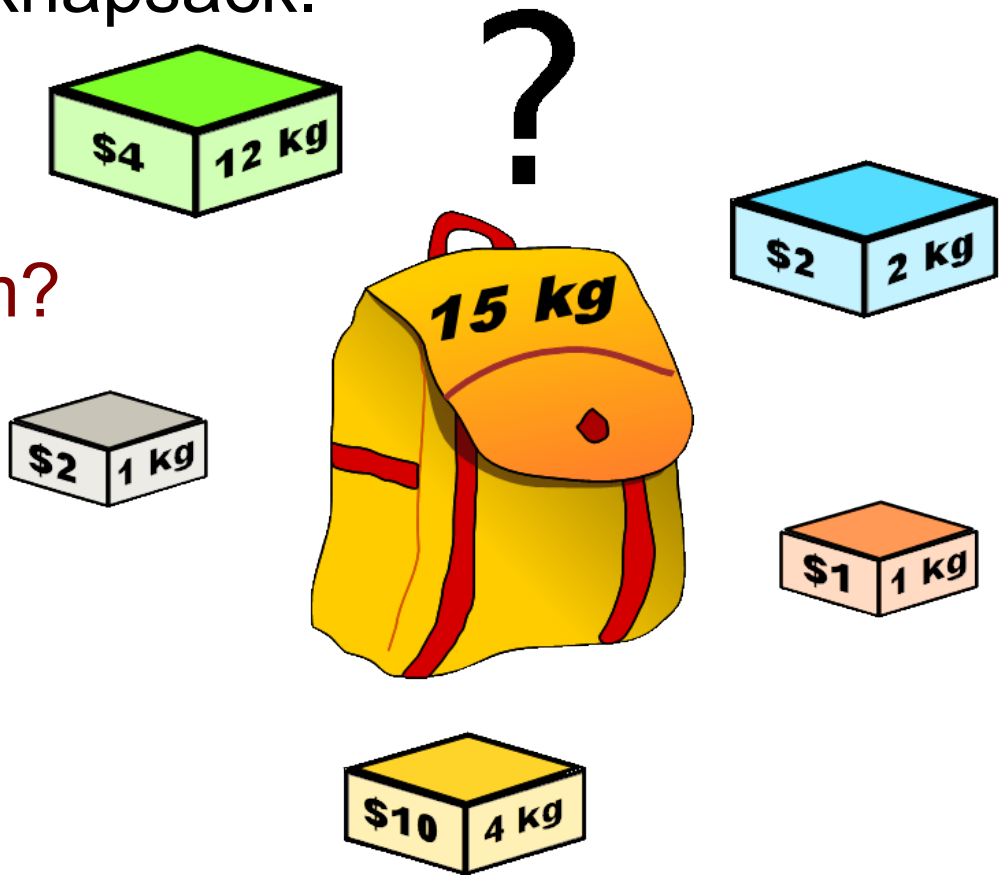
- S:={(v_1 ,w_1), (v_2 ,w_2), … (v_n ,w_n)}.

  (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

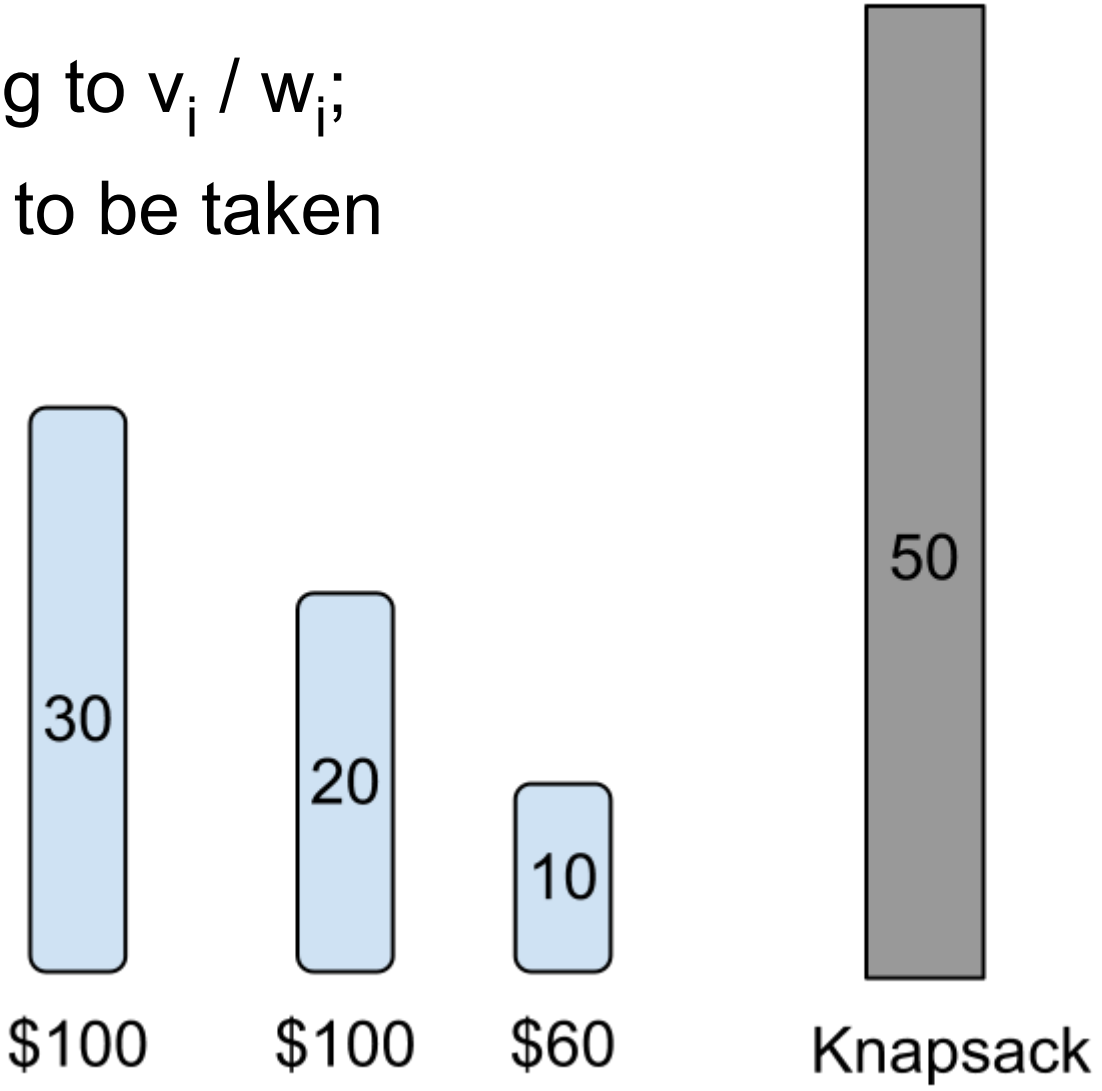
http://en.wikipedia.org/wiki/Knapsack_problem

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

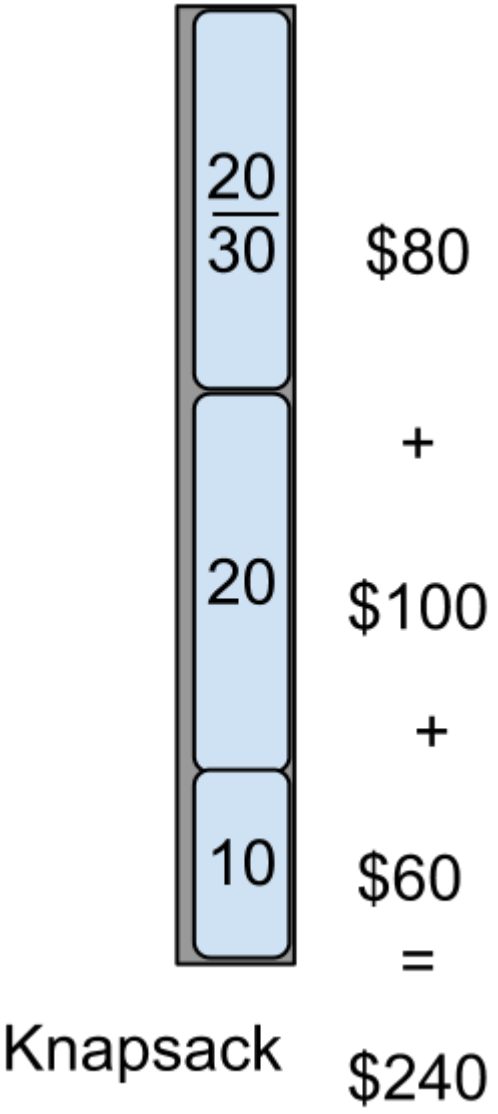
```
Fractional knapsack(W,S)
 Sort S, decreasingly according to v_i / w_i;
 x[1..n] = 0; //x[i] = amount of i to be taken
 weight = 0; i =1;

 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)
O(n log n)  {                     Sort S, decreasingly
                                  according to v_i / w_i;
O(n)        {                     for(i =1; i ≤ n; i++)
                                      x[i] =0;
                                  Weight = 0; i =1;
O(n)        {                     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
```

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