


EECE7205: Fundamentals of Computer Engineering



Greedy Algorithms



Introduction

- A ***greedy algorithm*** always makes the choice that looks best at the moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Greedy algorithms do not always yield optimal solutions, but for many problems they do.
- The greedy method is quite powerful and works well for a wide range of problems.
- *Minimum-spanning-tree* algorithms furnish a classic example of the greedy method.



An Activity Selection Problem

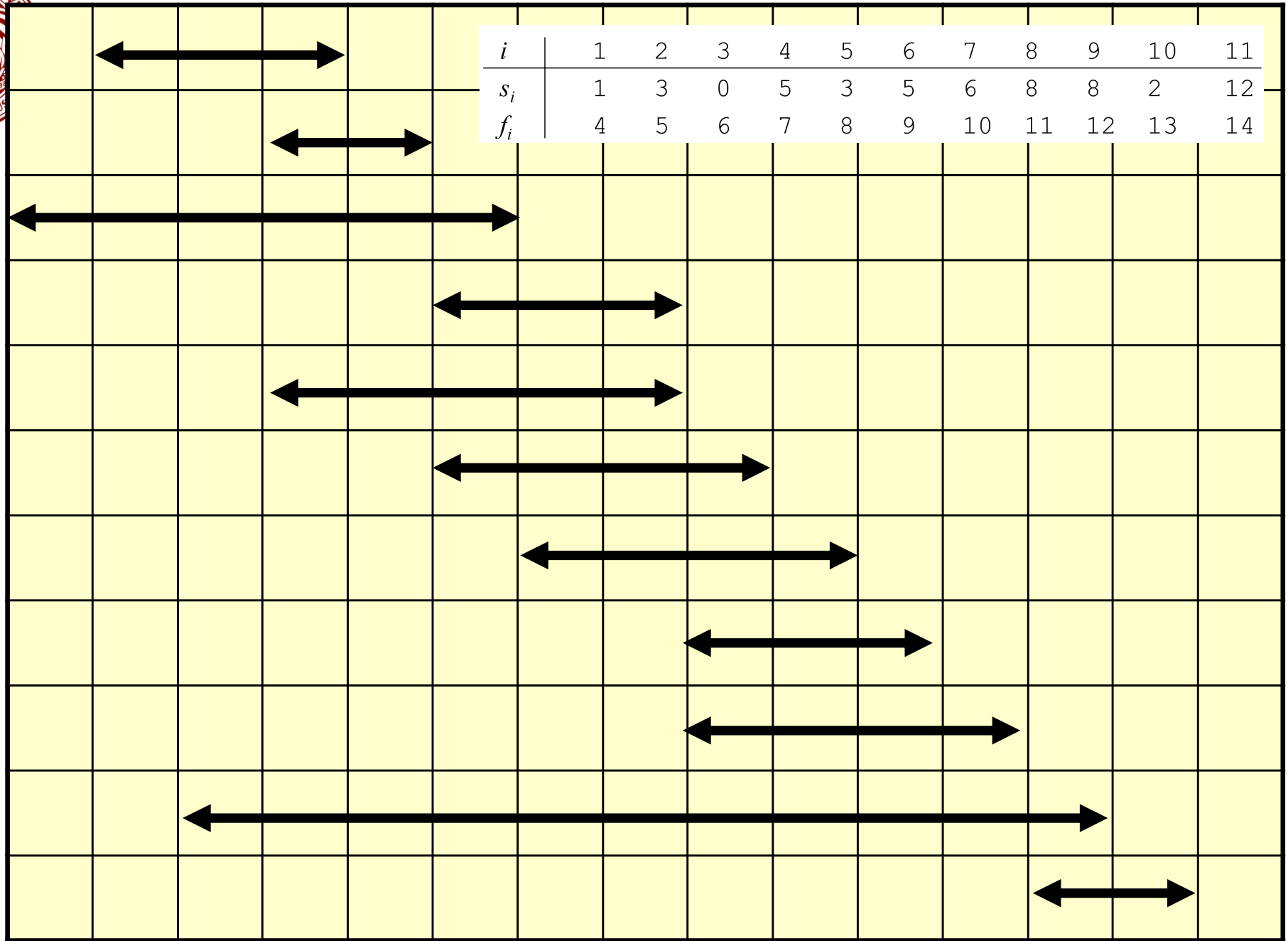
- *Input:* A set of activities $S = \{a_1, \dots, a_n\}$.
- Each activity has start time and a finish time, $a_i = (s_i, f_i)$.
- Two activities are compatible iff their interval does not overlap.
- *Output:* a maximum-size subset of mutually compatible activities.
- Example of S :

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

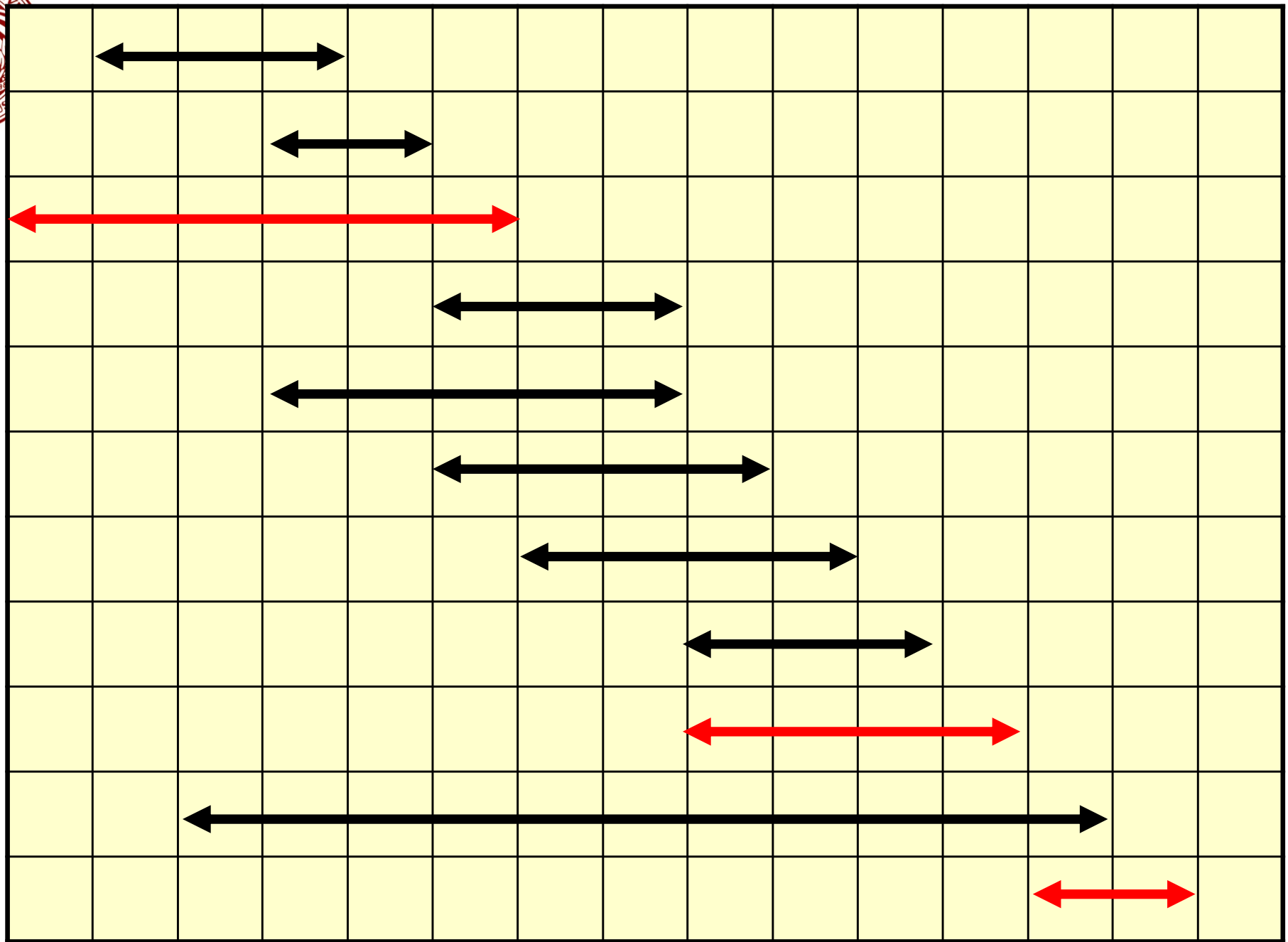
- Possible solutions:
 - $\{a_3, a_9, a_{11}\}$ can be completed.
 - But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set.
 - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$.



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

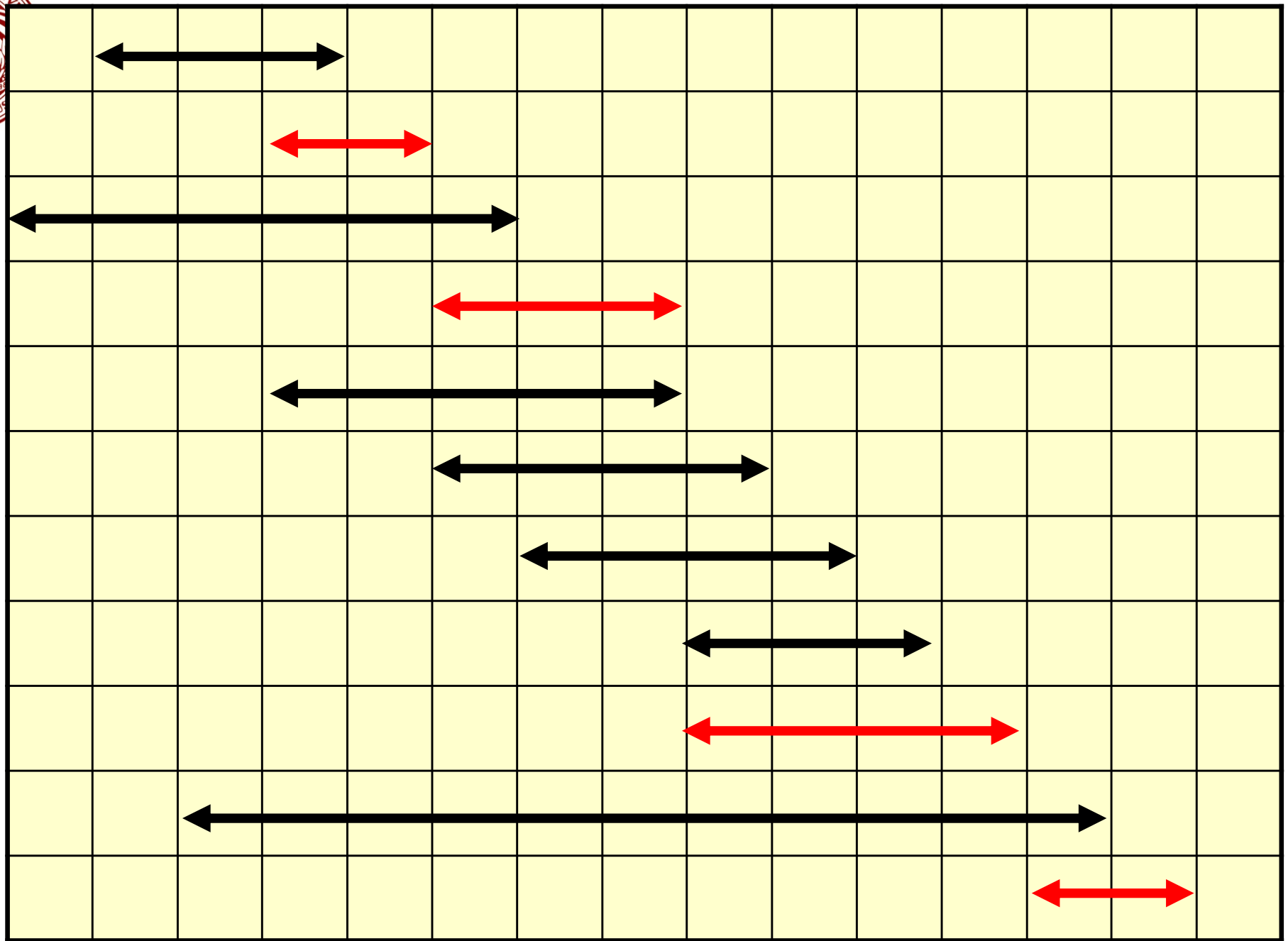


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



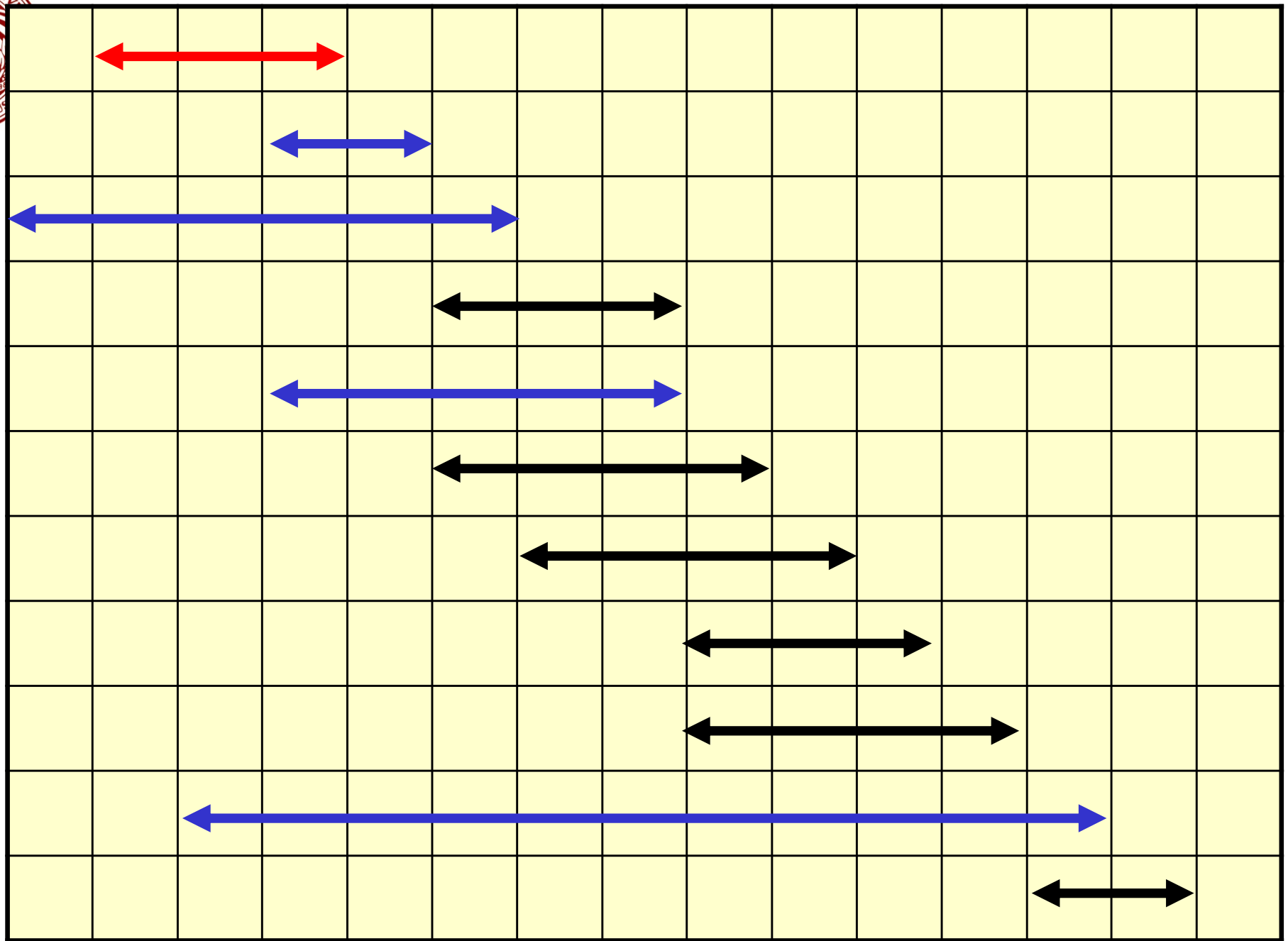


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

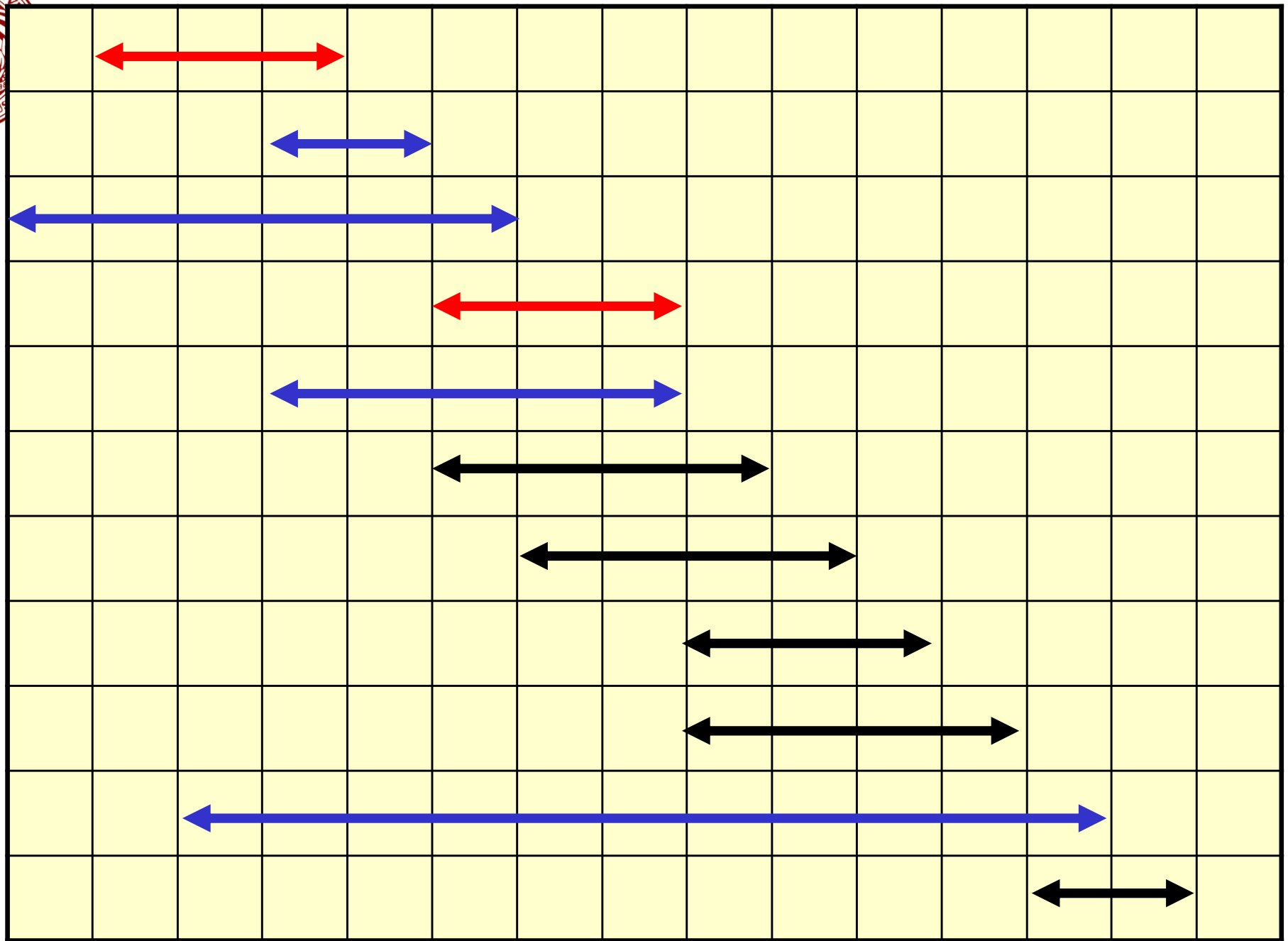


Early Finish Greedy Approach

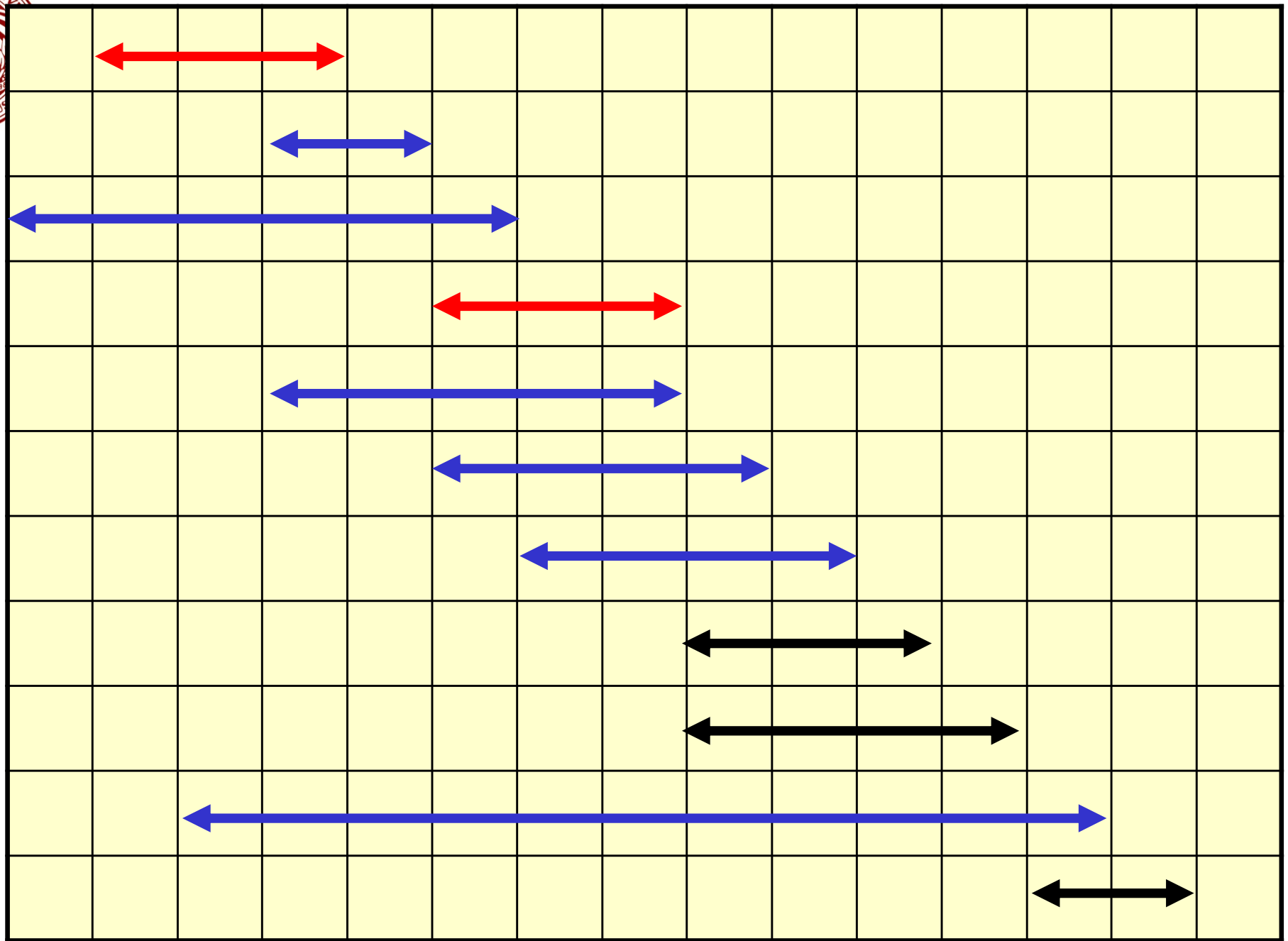
- Select the activity with the earliest finish.
- Eliminate the activities that could not be scheduled.
- Repeat!



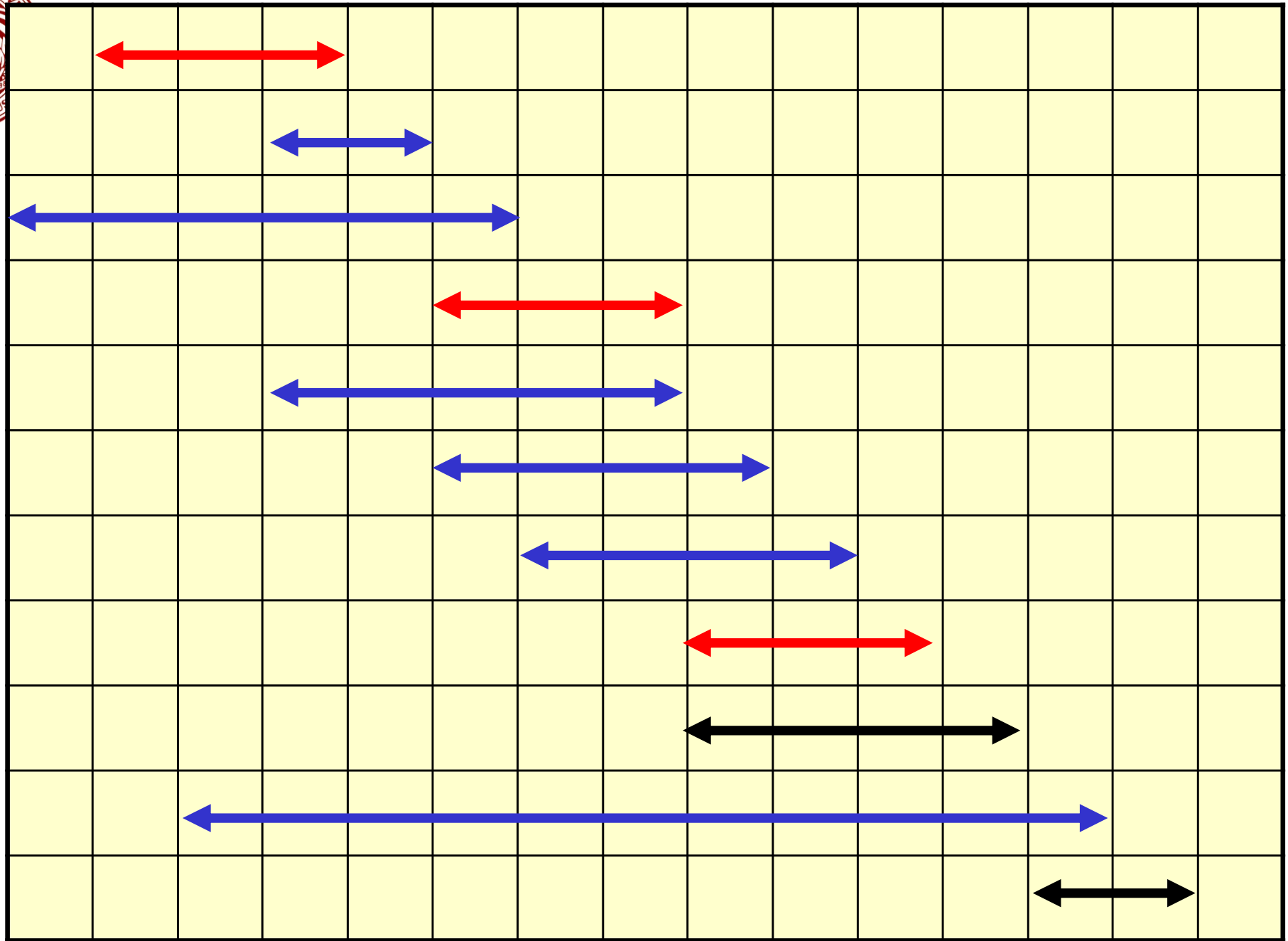
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



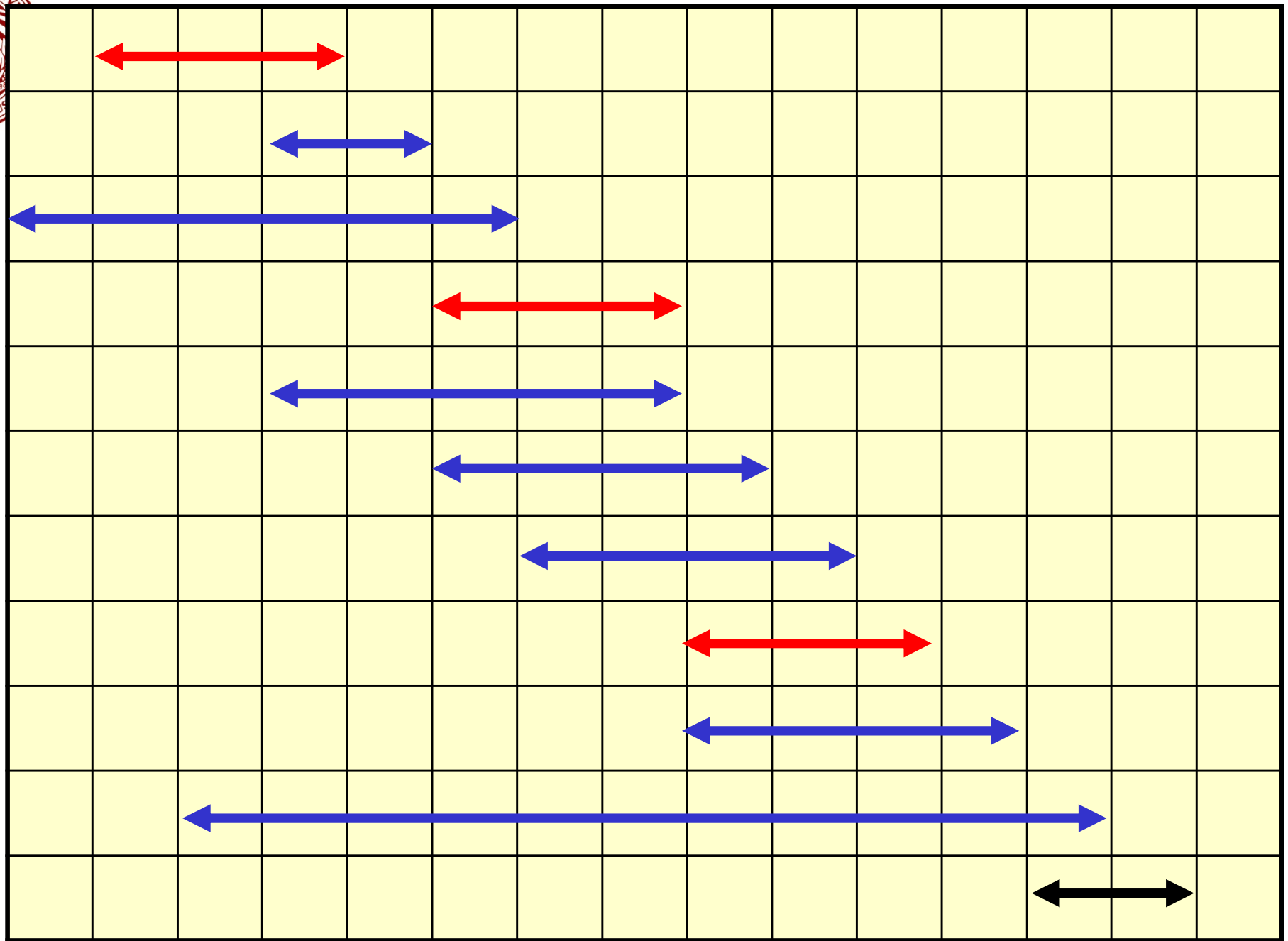
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



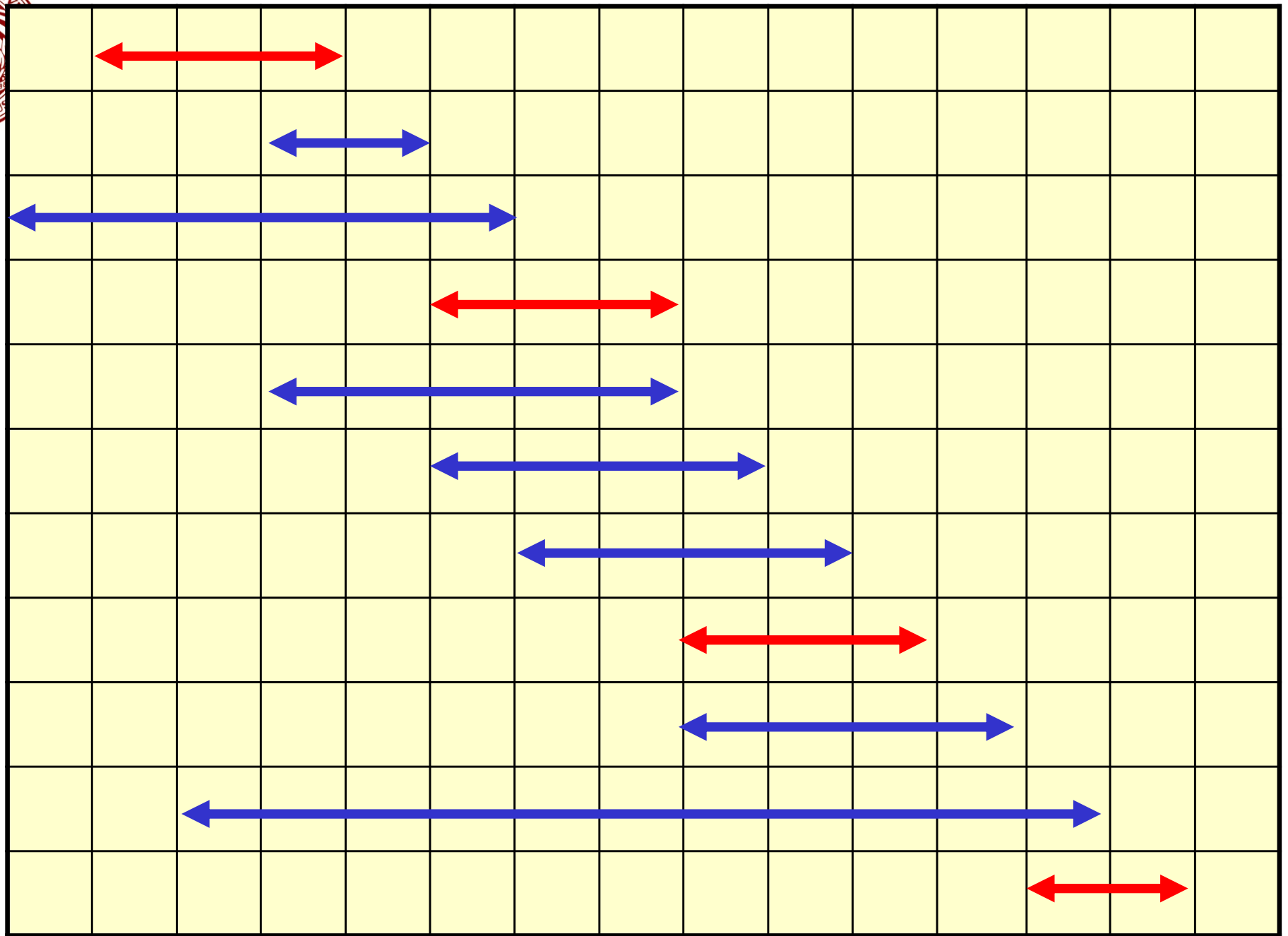
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



Greedy Activity Selector

- The procedure GREEDY-ACTIVITY-SELECTOR assumes that the input activities are **ordered by increasing finish time** in input f while input s is the array of their corresponding start time.
- It collects selected activities into a set A and returns this set when it is done.
- It is **greedy** in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled.
- The **greedy** choice is the one that maximizes the amount of unscheduled time remaining.

GREEDY-ACTIVITY-SELECTOR(s, f)

```

1   $n = s.length$ 
2   $A = \{a_1\}$ 
3   $k = 1$ 
4  for  $m = 2$  to  $n$ 
5      if  $s[m] \geq f[k]$ 
6           $A = A \cup \{a_m\}$ 
7           $k = m$ 
8  return  $A$ 
  
```




Elements of Greedy Strategy

- A greedy algorithm makes a sequence of choices, each of the choices that seems best at the moment is chosen.
 - Not always produce an optimal solution.
- Two ingredients that are exhibited by most problems that lend themselves to a greedy strategy.
 - Greedy-choice property.
 - Optimal substructure.



Greedy-Choice Property

- A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
 - Make whatever choice seems best at the moment and then solve the sub-problem arising after the choice is made.
 - The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems.
- We must prove that a greedy choice at each step yields a globally optimal solution.



Optimal Substructures

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub-problems.
 - If an optimal solution A to S begins with activity 1, then $A' = A - \{1\}$ is optimal to $S' = \{i \in S: s_i \geq f_1\}$
 - Why?
 - If we could find a solution B' to S' with more activities than A' , adding activity 1 to B' would yield a solution B to S with more activities than A → **contradicting the optimality of A .**
- After each greedy choice is made, we are left with an optimization problem of the same form as the original problem.
 - By induction on the number of choices made, making the greedy choice at every step produces an optimal solution.



The Knapsack Problem

- One wants to pack n items in a luggage.
 - The i^{th} item is worth v_i dollars and weighs w_i pounds.
 - Maximize the value but cannot exceed W pounds.
 - v_i, w_i, W are integers.
- 0-1 knapsack \rightarrow each item is taken or not taken.
- Fractional knapsack \rightarrow fractions of items can be taken.
- Both exhibit the optimal-substructure property.
 - **0-1:** If item j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most $W - w_j$.
 - **Fractional:** If w pounds of item j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most $W - w$ that can be taken from other $n-1$ items plus $w_j - w$ of item j .



Fractional Knapsack Greedy Algorithm

- Fractional knapsack can be solvable by the greedy strategy
 - Compute the value per pound v_i / w_i for each item
 - Following a greedy strategy, take as much as possible of the item with the greatest value per pound.
 - If the supply of that item is exhausted and there is still more room, take as much as possible of the item with the next value per pound, and so forth until there is no more room.

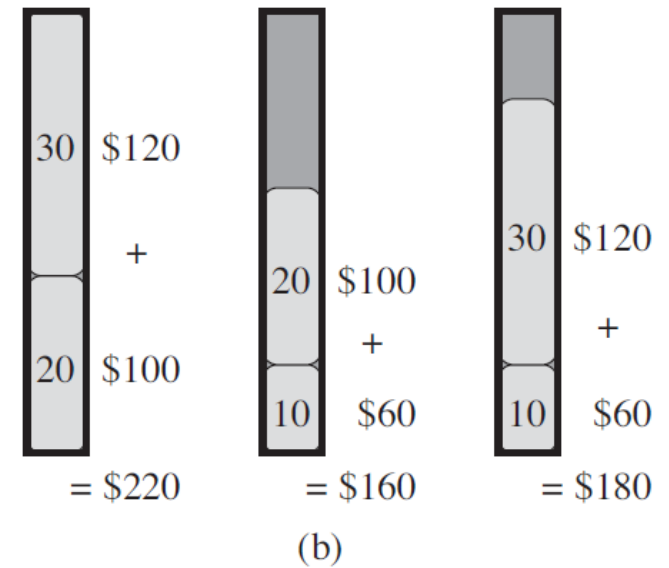
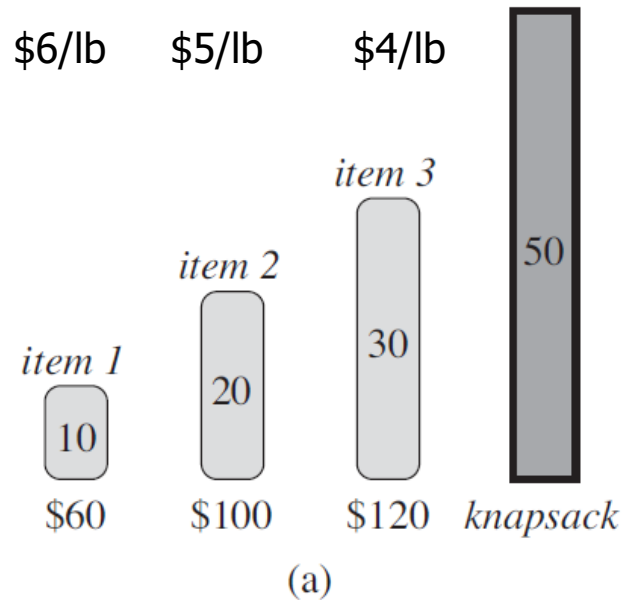


0-1 Knapsack is Harder!

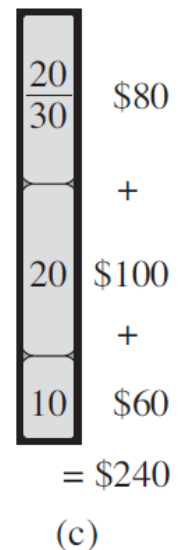
- 0-1 knapsack cannot be solved by the greedy strategy. Why?
 - Unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the current packing.
 - We must compare the solution to the sub-problem in which the item is included with the solution to the sub-problem in which the item is excluded before we can make the choice.
 - **Dynamic Programming**



Knapsack Example



- The traveller must select a subset of the three items shown whose weight must not exceed 50 pounds.
- The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound.
- For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.





Data Compression

- Suppose we have 1000,000,000 (1G) character data file
- Suppose the file only contains 26 letters $\{a, \dots, z\}$.
- Suppose each letter α in $\{a, \dots, z\}$ occurs with frequency f_α
- Suppose we encode each letter by a binary code.
 - If we use a fixed length code, we need 5 (as $2^5 = 32$) bits for each character
 - The resulting message length is $5(f_a + f_b + \dots + f_z)$
- **Can we do better?**



Data Compression (Cont'd)

- Most character code systems (ASCII, unicode) use fixed length encoding.
- If frequency data is available and there is a wide variety of frequencies, variable length encoding can save 20% to 90% space.
- Which characters should we assign shorter codes; which characters will have longer codes?



Data Compression Example

- Suppose the file only has 6 letters {a,b,c,d,e,f} with frequencies (percentage of occurrence)

<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	
.45	.13	.12	.16	.09	.05	
000	001	010	011	100	101	Fixed length
0	101	100	111	1101	1100	Variable length

- Fixed length 3G=3000,000,000 bits
- Variable length:

$$(.45 \bullet 1 + .13 \bullet 3 + .12 \bullet 3 + .16 \bullet 3 + .09 \bullet 4 + .05 \bullet 4) = 2.24G$$
- Space saving of about 25%.



Variable Length Encoding Requirements

- No encoding of a character can be the prefix of the longer encoding of another character, for example, we could not encode t as 01 and x as 01101 since 01 is a prefix of 01101.
- Prefix codes allow easy decoding
 - Given a: 0, b: 101, c: 100, d: 111, e: 1101, f: 1100
 - Decode 001011101 going left to right:

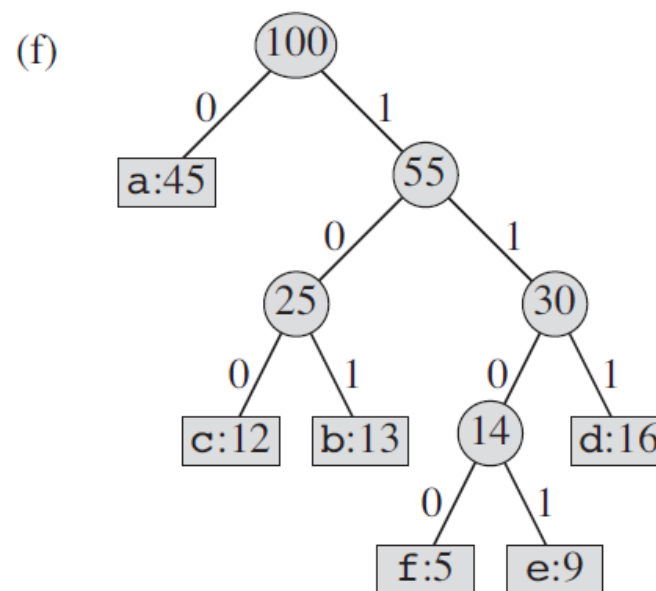
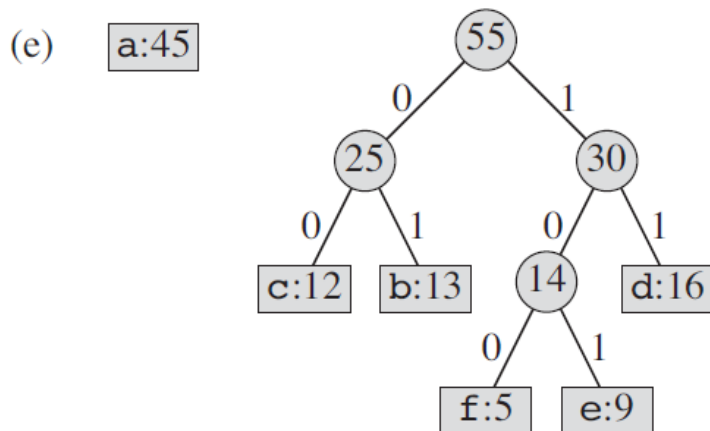
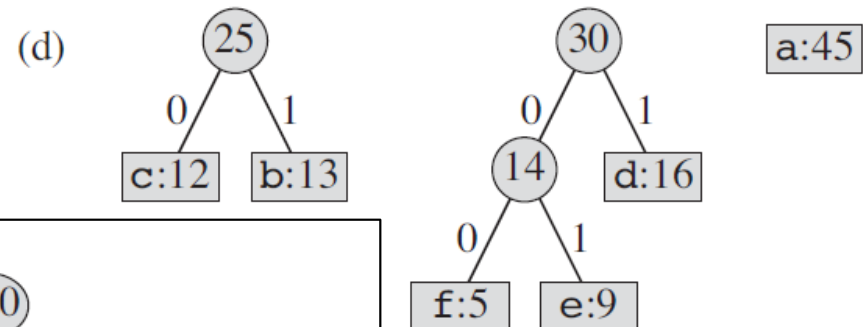
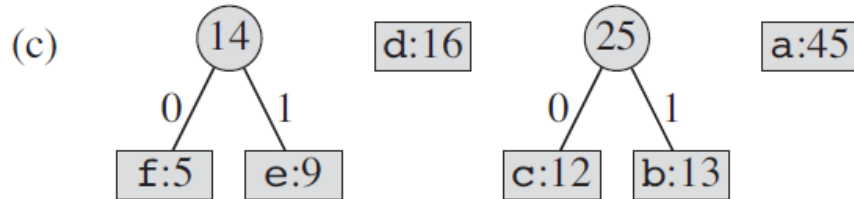
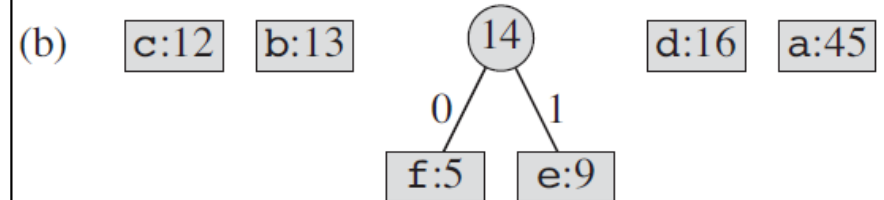

```
0|01011101
a|0|1011101
a|a|101|1101
a|a|b|1101
a|a|b|e
```



Building the Encoding Tree

- At each step, the two trees with lowest frequencies are merged.

(a) f:5 e:9 c:12 b:13 d:16 a:45



a	0
b	101
c	100
d	111
e	1101
f	1100



Building the Encoding Tree (Cont'd)

- In the tree, 0 means “go to the left child” and 1 means “go to the right child.”
- We interpret the binary codeword for a character as the simple path from the root to that character.
- For each character c in the alphabet C , let the attribute $c.freq$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c 's leaf in the tree.
 - Note that $d_T(c)$ is also the length of the codeword for character c .
- The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} c.freq \times d_T(c)$$



The Huffman Algorithm

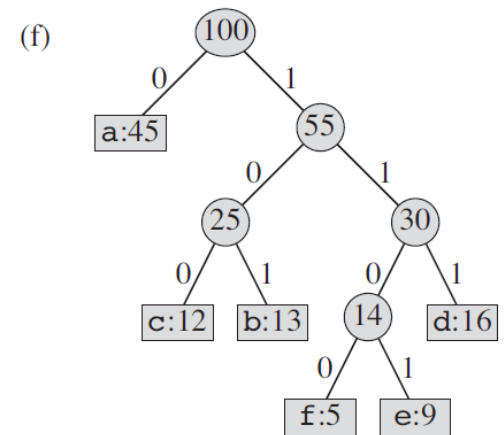
- Huffman's greedy algorithm builds the prefix encoding tree by taking as input the set C of n characters. Each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency.
- The algorithm uses a min-priority queue Q , keyed on the $freq$ attribute, to identify the two least-frequent objects to merge.
- After $n-1$ mergers, line 9 returns the one node left in the queue, which is the root of the code tree.

HUFFMAN(C)

```

1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8       $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$     // return the root of the tree

```



Switching the left and right child (x and y) of any node yields a different code of the same cost.