COMP 355
Advanced Algorithms

Finish Linear Time Sorting
Greedy Algorithms for Scheduling

Rhodes College
—1848—

Linear-Time Sorting

- The $\Omega(n \log n)$ lower bound implies that if we hope to sort numbers faster than in $O(n \log n)$ time, we cannot do it by making comparisons alone.
- **Counting Sort**: assumes each integer in range from 1 to k.
- **Radix Sort**: only practical for very small ranges of integers.
- **BucketSort**: works for floating-point numbers, but should only be used if numbers are roughly uniformly distributed over some range.

BucketSort

```
BUCKET_SORT(A):
    n = A.length
    let B[0…n-1] be a new array
    for i = 0 to n -1:
        make B[i] an empty list
    for i = 0 to n:
        insert A[i] into list B[⌊nA[i]⌋]
    for i = 0 to n -1:
        sort list B[i] with insertion sort
    concatenate lists B[0], B[1],…, B[n-1] in order
```

Summary

Comparison-Based Sorting Algorithms: A *stable* sorting algorithm preserves the relative order of equal elements. An *in-place* sorting algorithm uses no additional array storage (although $O(\log n)$ additional space is allowed for the recursion stack).

Algorithm	Time	Stable	In-place
BubbleSort	$\Theta(n^2)$	Yes	Yes
InsertionSort	$\Theta(n^2)$	Yes	Yes
MergeSort	$\Theta(n \log n)$	Yes	No
HeapSort	$\Theta(n \log n)$	No	Yes
QuickSort*	$\Theta(n \log n)$	Yes/No	No/Yes

*There are two versions of QuickSort, one which is stable but not in-place, and one which is in-place but not stable.

Non-Comparison-Based Sorting Algorithms: All of these algorithms are stable, but not in-place.

Algorithm	Assumptions	Time	Space
CountingSort	Integers over $[0..k]$	$\Theta(n+k)$	$\Theta(n+k)$
RadixSort	Integers over $[0..n^d]$	$\Theta(d(n+k))$	$\Theta(n+k)$
BucketSort	Integers uniformly distributed	$\Theta(n)$ (Expected)	$\Theta(n)$

Questions

- Why is the worst-case running time of bucket sort $O(n^2)$? What simple change to the algorithm preserves its linear time average run-time and makes its worst-case running time $O(n \log n)$?

- Given the data set A = {6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2}, which sorting algorithm would you use?

- Show how to sort n integers in the range 0 to n^3-1 in $\Theta(n)$ time.

Greedy Algorithms

- **Def:** Algorithms that make locally optimal choices using a metric with the hope of finding a globally optimal solution.
- **Example**: Making change with US coins.
- **Optimization Problem**: Given an input, compute a solution, subject to various constraints, that either minimizes cost or maximizes profit.

Coin-Changing: Greedy Algorithm

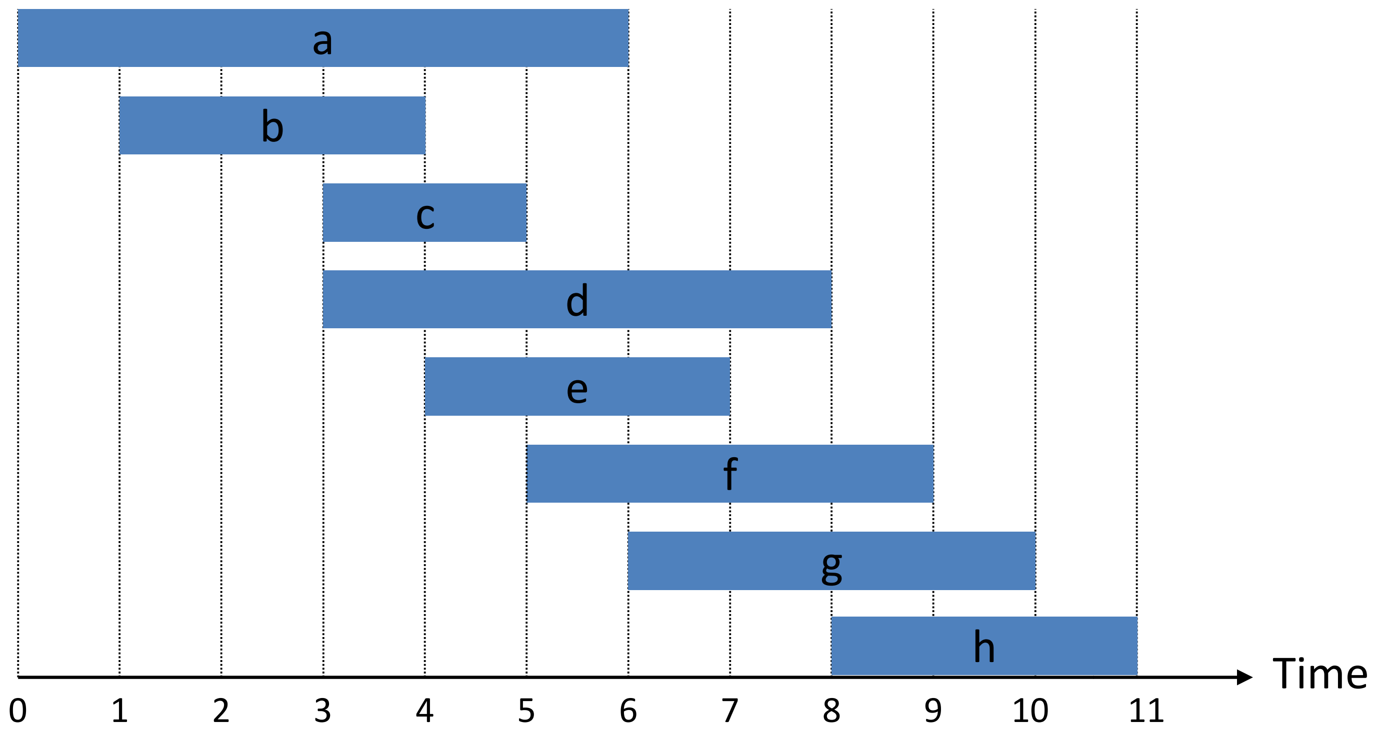
Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

```
Sort coins denominations by value: c₁ < c₂ < … < cₙ.

    coins selected
S ← φ
while (x ≠ 0) {
   let k be largest integer such that cₖ ≤ x
   if (k = 0)
      return "no solution found"
   x ← x - cₖ
   S ← S ∪ {k}
}
return S
```

Interval Scheduling

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: Algorithm

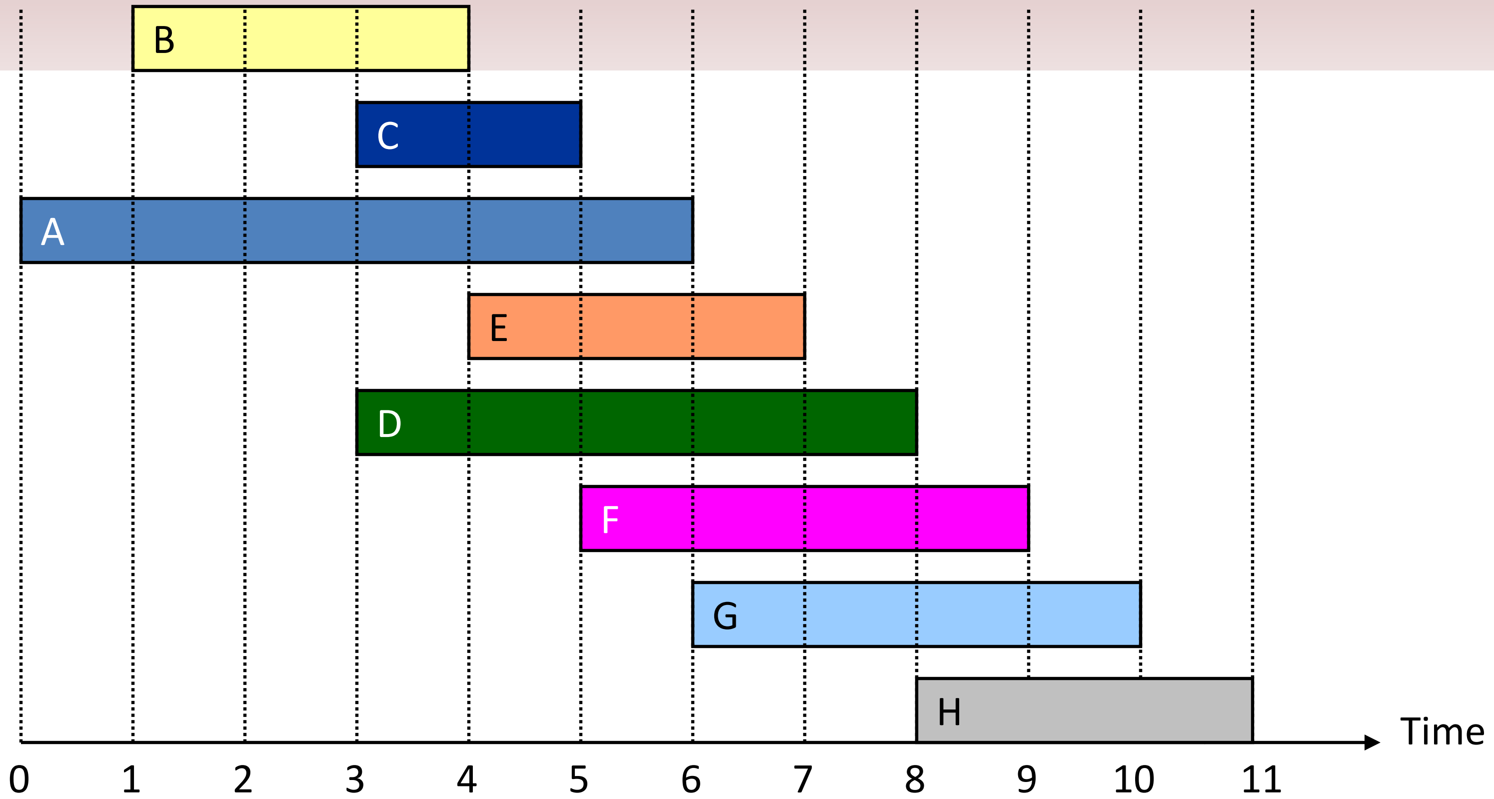
Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

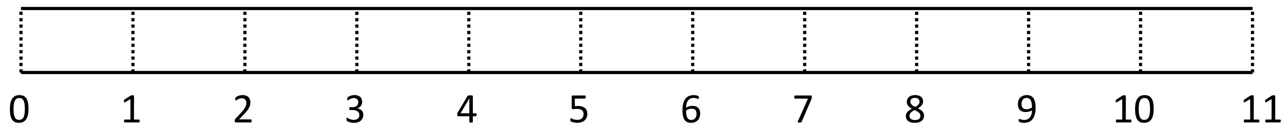
```
Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ.

     jobs selected
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation. O(n log n).

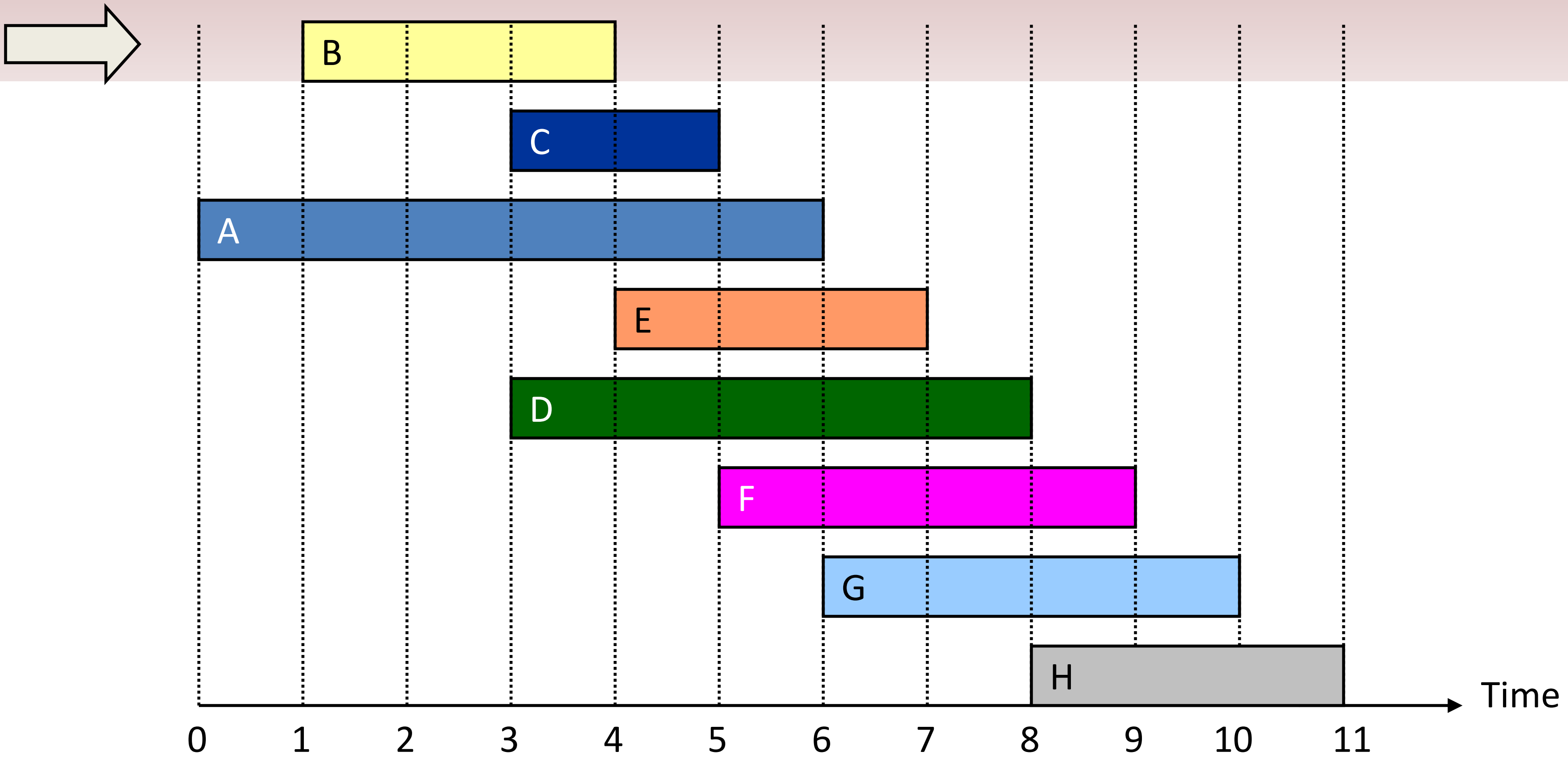
- Remember job j* that was added last to A.
- Job j is compatible with A if $s_j \geq f_{j^*}$.

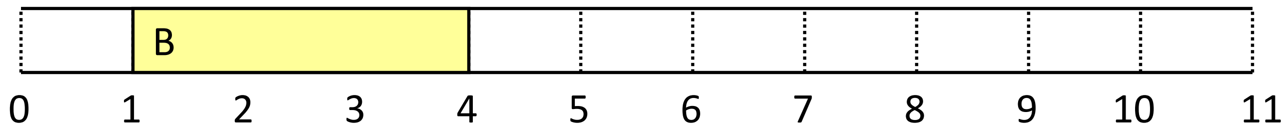
Interval Scheduling



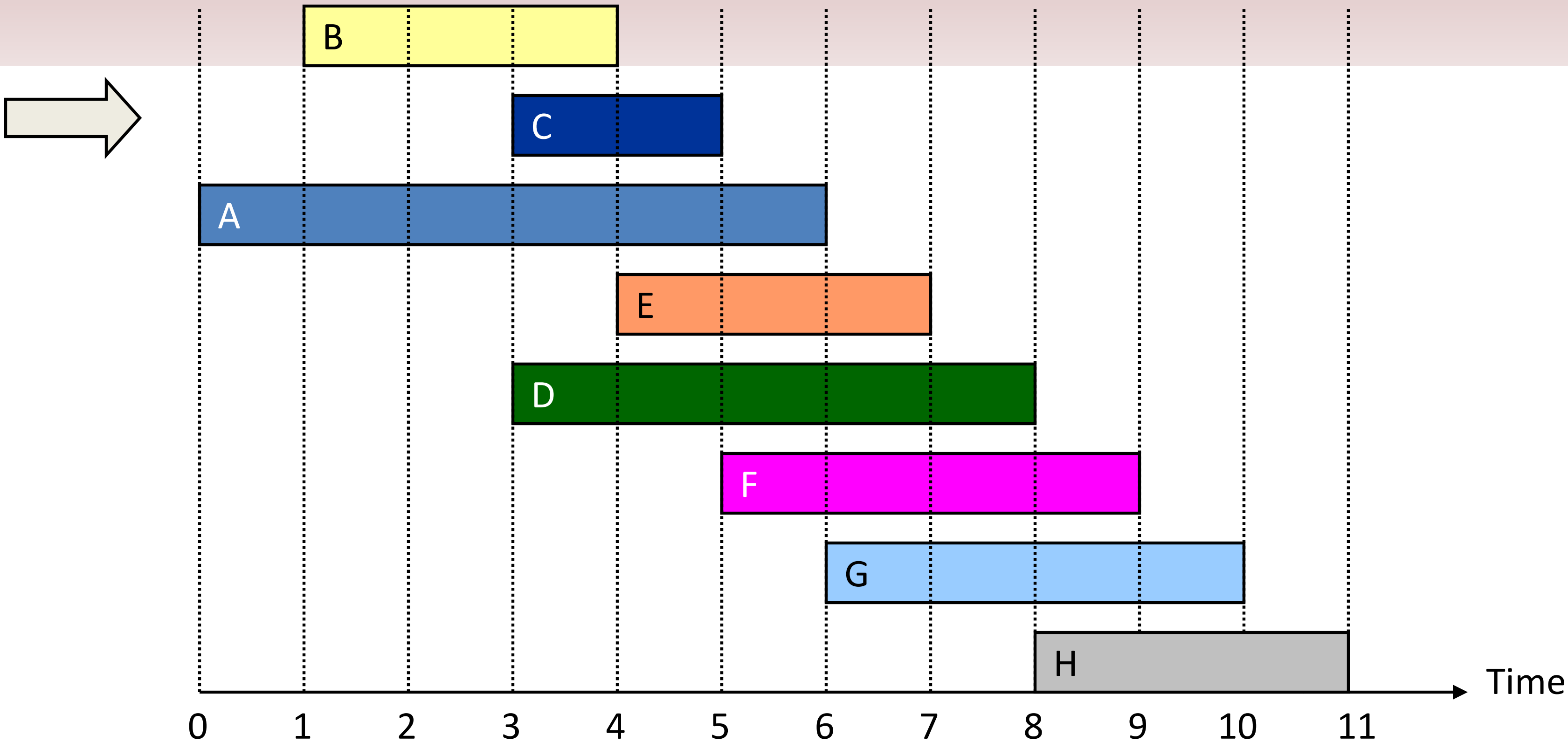


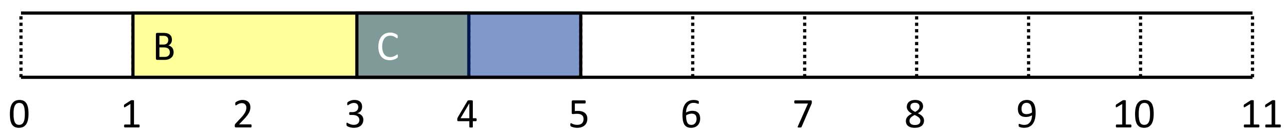
Interval Scheduling



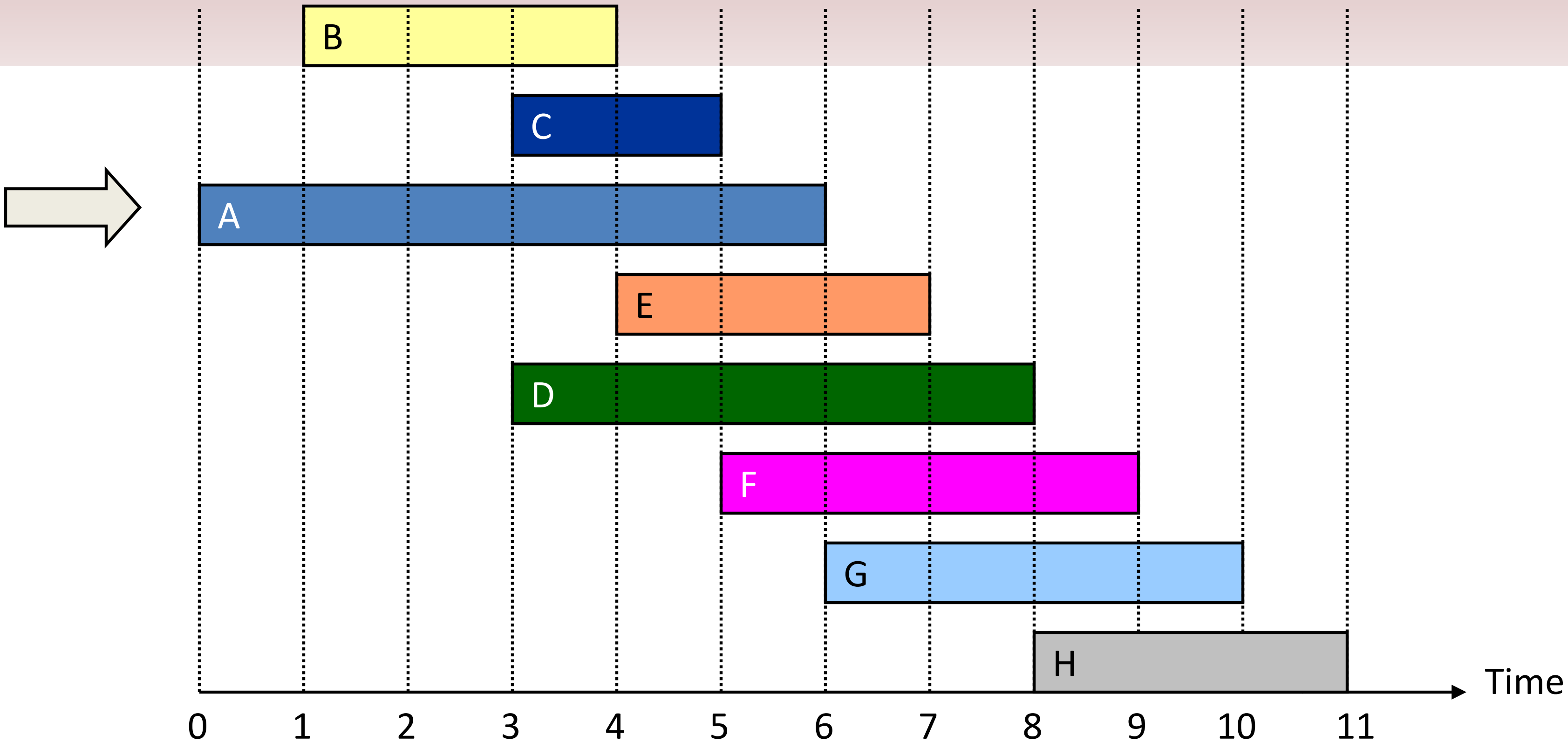


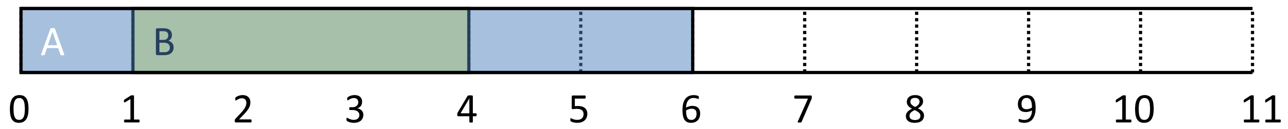
Interval Scheduling



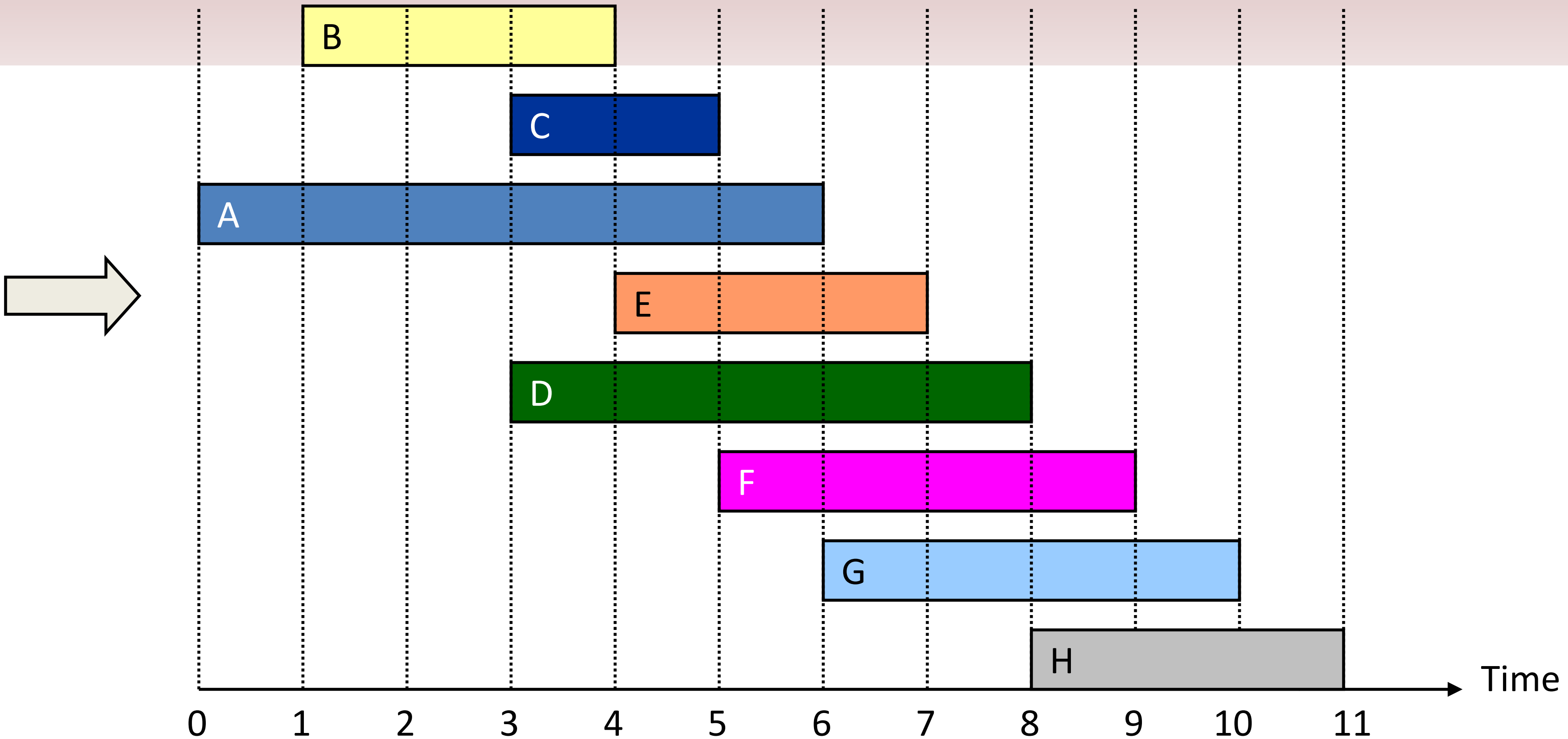


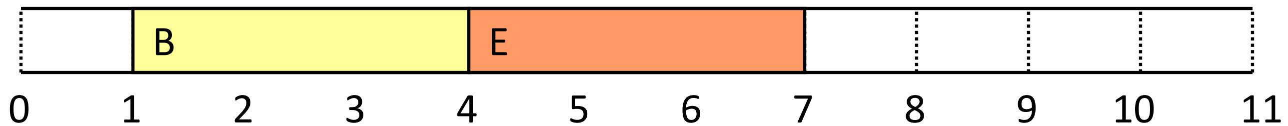
Interval Scheduling



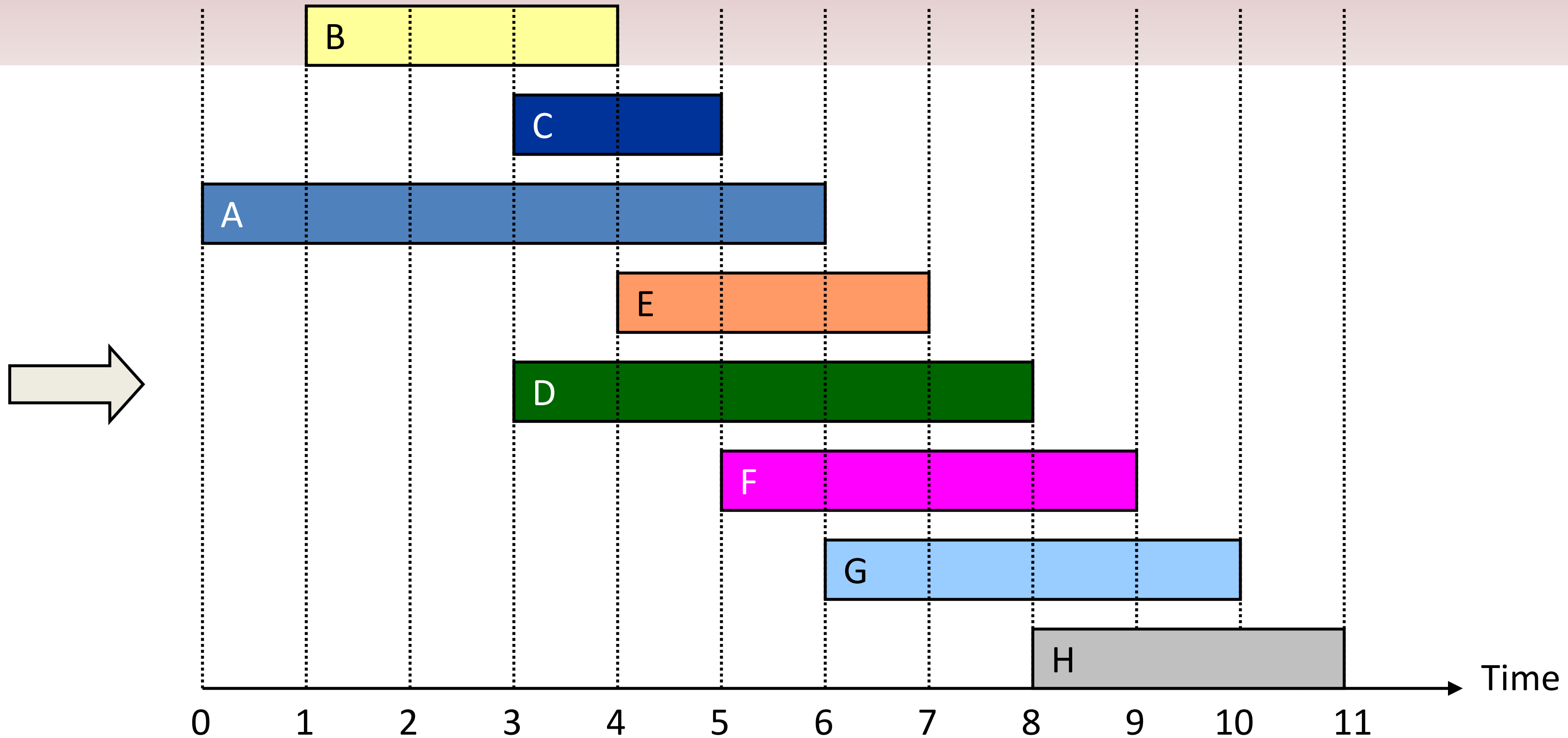


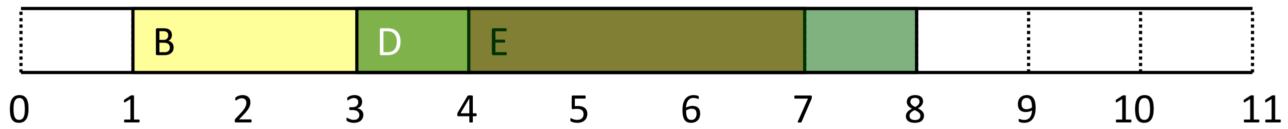
Interval Scheduling



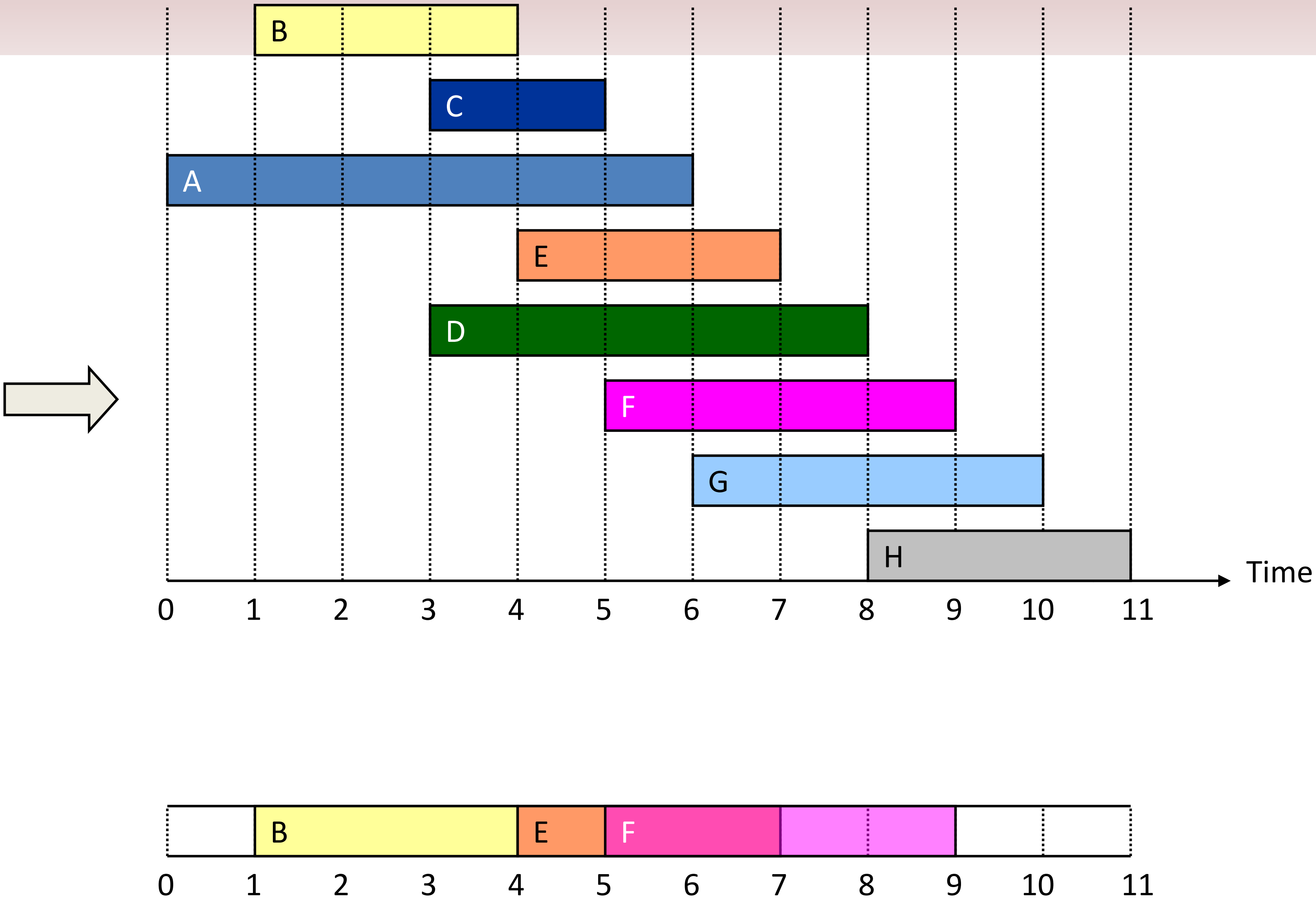


Interval Scheduling

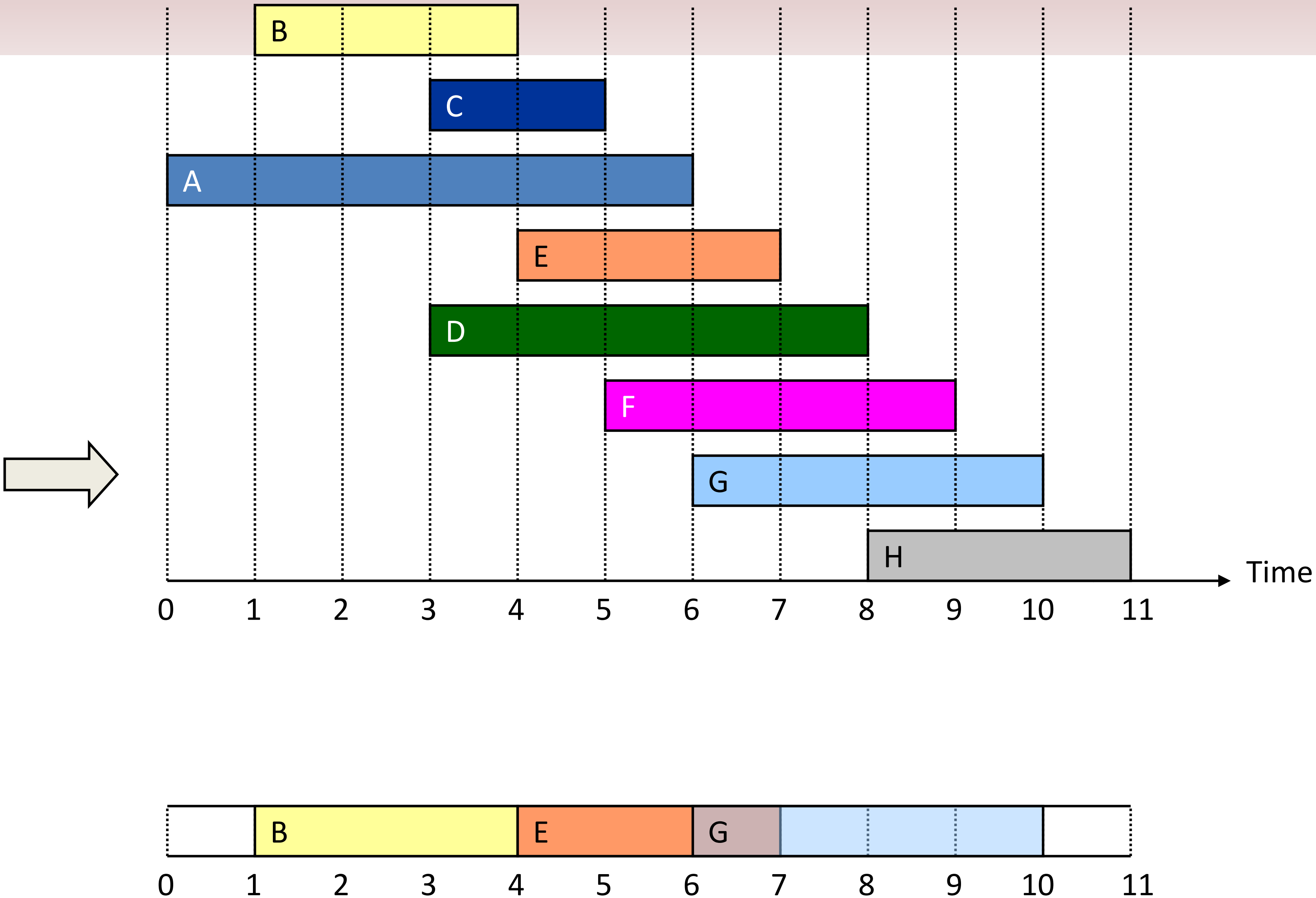




Interval Scheduling



Interval Scheduling



Interval Scheduling