CSE 6140/ CX 4140:

Computational Science and Engineering ALGORITHMS

Instructor: Anne Benoit

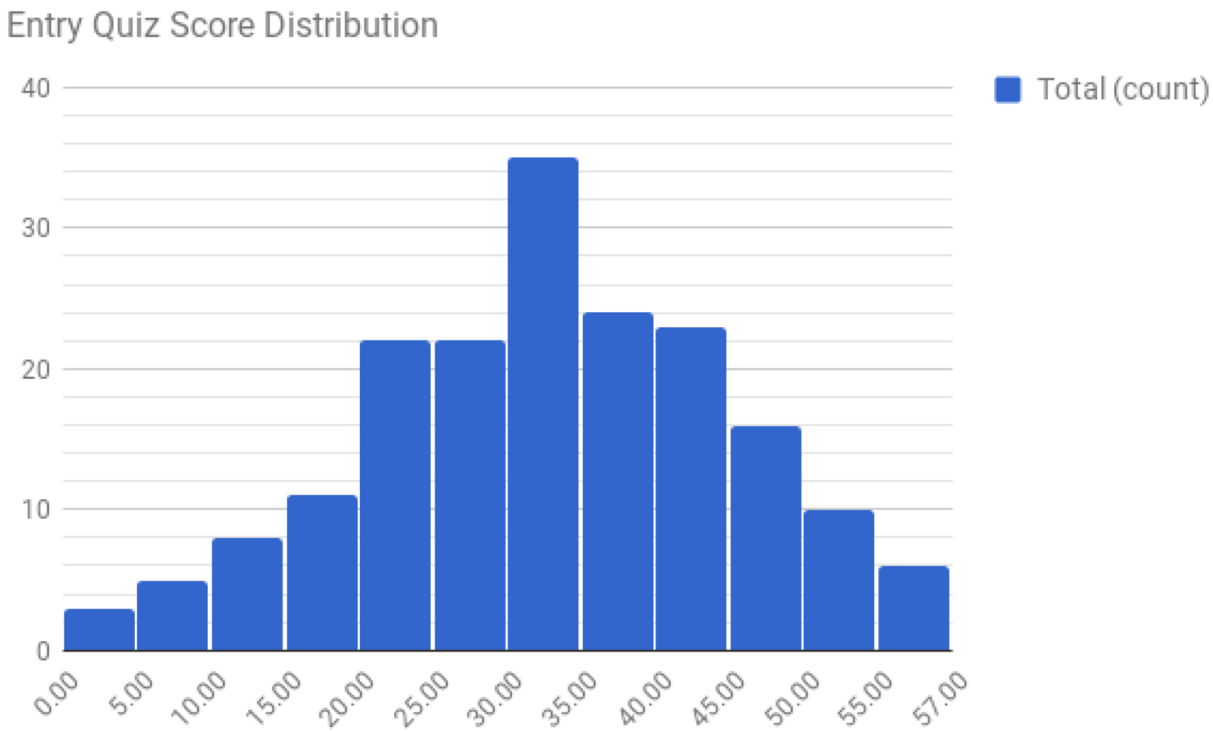
Visiting Associate Professor, CSE

Based on slides by Bistra Dilkina

Entry Quiz

- Mean: 32.6 (35.2 last year) (between 2 and 57)

- Q1: 4.3/7
- Q2: 1.6/4
- Q3: 2.7/6
- Q4: 6/10
- Q5: 11.9/15
- Q6: 2.4/5
- Q7: 3.6/10



Schedule

- Aug 22: Introduction and Entry quiz. Reading: review material from quiz if you were not confident, for instance read KT1, KT2, KT3.

- Aug 24: Review of proofs and Greedy algorithms. Reading: KT2.1, KT2.2, KT1.2, KT4.1.

- Aug 29: Greedy algorithms (interval scheduling). Reading: KT4.1, KT4.2.
- Aug 31: Greedy algorithms (graphs, shortests paths). Reading: KT4.3, KT4.4.

The kind of problems

- Design an algorithm such that
- Given any valid input
  - e.g. a graph, a set of strings from an alphabet, a set of jobs

- Find an output that:
- Satisfies a set of requirements (feasible)
  - e.g. a set of vertices that are not neighbors in the graph
  - e.g. a set of jobs that do not overlap

- And has the best possible quality based on a specified objective function (optimal)

Algorithm

- Definition
  - A sequence of computational steps that transform the input into a desired output

- Proof of correctness
  - On every valid input, the algorithm produces the output that satisfies the required input/output relationship
    - Review of proofs

- Analysis of time and space
  - Given the ‘size’ of the input, how many computational steps does the algorithm take?
    - Review of Asymptotic Notations
    - Review of Recurrences

REVIEW OF PROOFS

Proofs by counterexample

Proofs by contradiction

Inductive proofs

If-and-only-if proofs

Etc.

Questions 3 & 4 of Quiz in class

Why algorithm design matters?

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

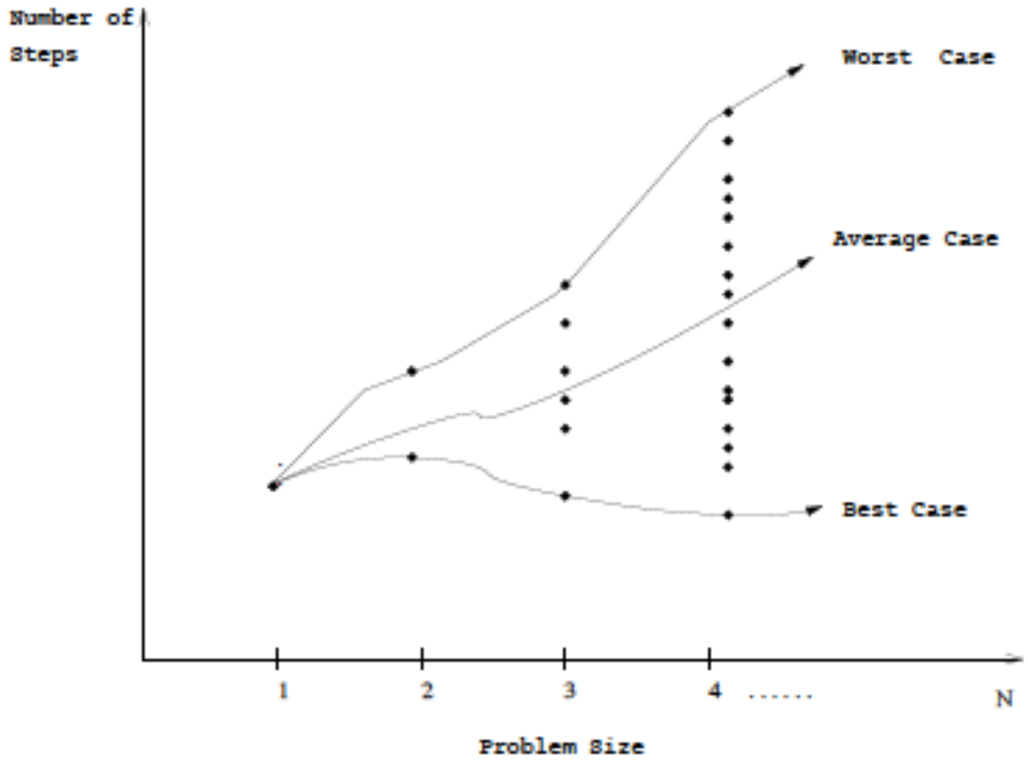
	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Running time

- Model of computation: Random-Access Machine (RAM)
  - Instructions are executed one after the other (non concurrency)
  - Each "simple" operation takes 1 step: (+, -, =, if, call, memory access)
  - Loops and subroutine calls are *not* simple operations. They depend upon the size of the data and the contents of a subroutine. "Sort" is not a single step operation
- Input size
  - Depends on the problem being studied
  - E.g., number of items in the input
  - E.g., number of nodes and number of edges in a graph input
- Running time
  - Number of steps taken by the algorithm, given input size

Running time

- Worst-case: the maximum number of steps taken on any instance of size n
- Best-case: the minimum number of steps taken on any instance of size n.
- Average-case: the average number of steps taken on any instance of size n. (Need an understanding of the distribution of possible inputs)

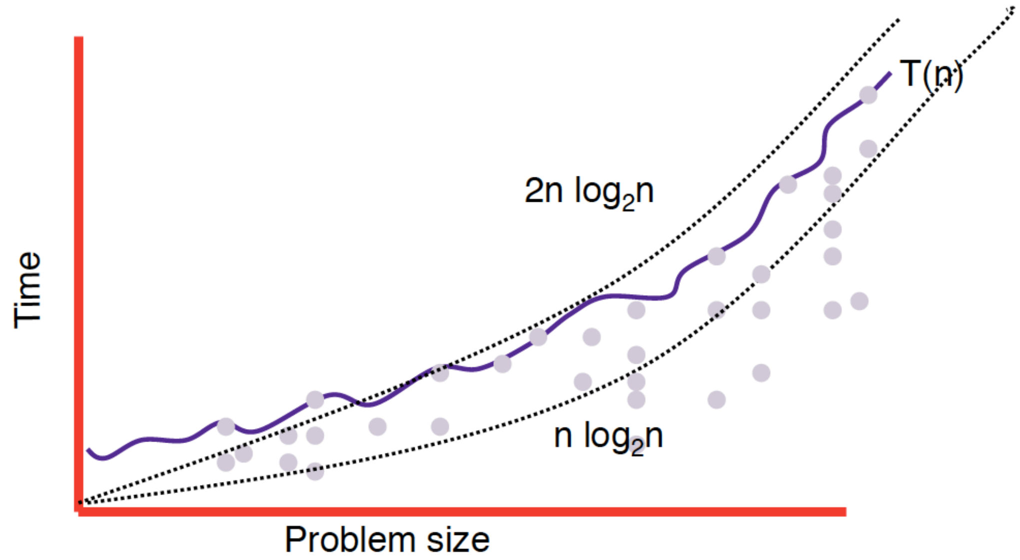


Time complexity

- The *time complexity* of an algorithm associates a number T(n), the maximum (worst-case) amount of time taken on any input of size n
- Mathematically, T: N → R
  i.e., T is a function mapping non-negative integers (problem sizes) to real numbers (number of steps).
  - "Reals" so we can say, e.g., sqrt(n) instead of ⌈sqrt(n)⌉
- A key question is "Scaling up" of the algorithm
  - What happens to my runtime if I have twice as many items, or twice as large graph?
  - (E.g. today: cn^2, next year: $c(2n)^2 = 4cn^2$: 4x longer.)
  - Big-Oh notation answers this kind of questions

Exact analysis is hard

- T(n) might be difficult to deal with precisely because the details are very complicated:



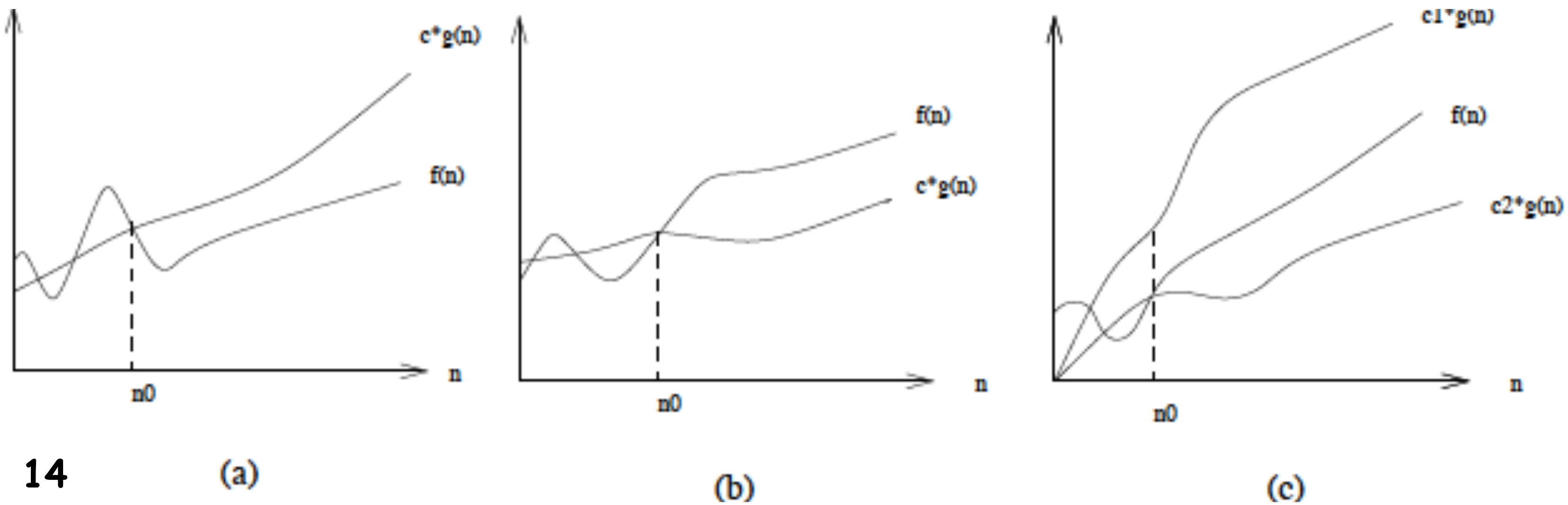
- It easier to talk about *upper and lower bounds* of the function
- Asymptotic notation (O, Ω, and Θ) are the way we practically deal with analysis of time complexity

Big-Oh notation

ASYMPTOTIC ORDER OF GROWTH

Asymptotic Order of Growth

- **Upper bounds**: $T(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot g(n)$.
- **Lower bounds**: $T(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot g(n)$.
- **Tight bounds**: $T(n)$ is $\Theta(g(n))$ if $T(n)$ is both $O(g(n))$ and $\Omega(g(n))$.



Notation

- Example: $T(n) = 32n^2 + 17n + 32$.
  - $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
  - $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.
- Slight abuse of notation. $T(n) = O(g(n))$
- **Equality not transitive**:
  - $T1(n) = 5n^3$; $T2(n) = 3n^2$
  - $T1(n) = O(n^3) = T2(n)$
  - But $T1(n) \neq T2(n)$.
- Better notation: $T(n) \in O(g(n))$
- T(n) is <u>in the set</u> of functions bounded from above by g(n)

Properties

- **Transitivity**
  - If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.
  - If $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$.
  - If $f \in \Theta(g)$ and $g \in \Theta(h)$ then $f \in \Theta(h)$.

- **Additivity**
  - If $f \in O(h)$ and $g \in O(h)$ then $f + g \in O(h)$.
  - If $f \in \Omega(h)$ and $g \in \Omega(h)$ then $f + g \in \Omega(h)$.
  - If $f \in \Theta(h)$ and $g \in O(h)$ then $f + g \in \Theta(h)$.

Some Common Functions

- Polynomials: $a_0 + a_1 n + \dots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

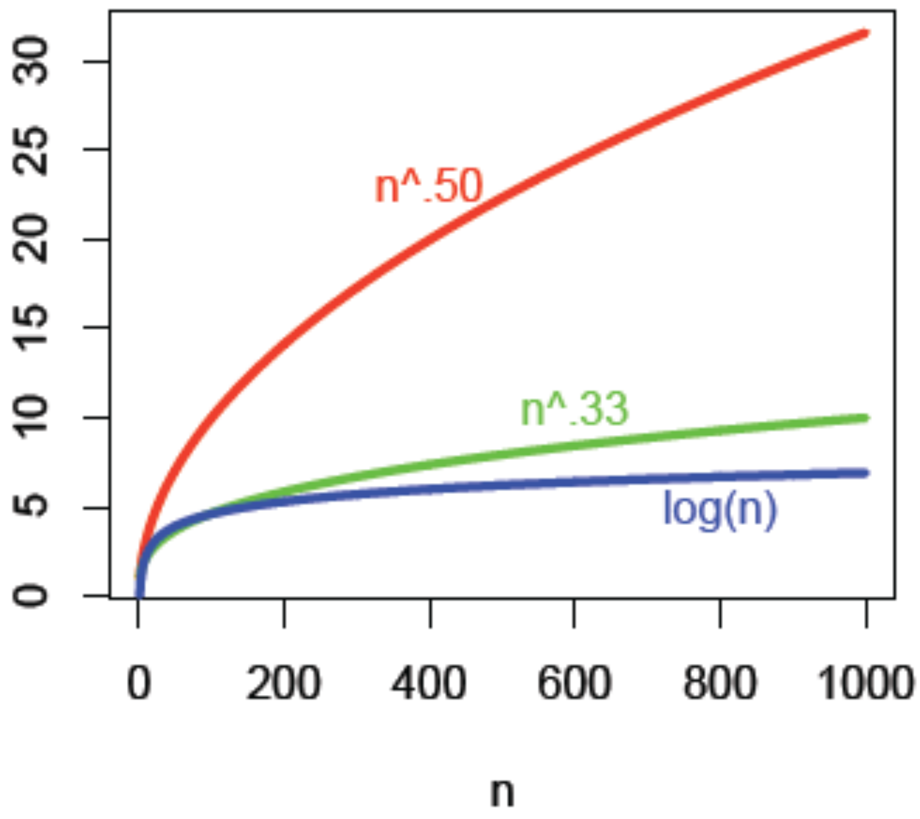
- Logarithms: the base does not matter
  - $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$.
  - Recall $\log_b n = \log_a n / \log_a b$

- Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

- Exponential time. Running time is $O(r^n)$ for some constant r.

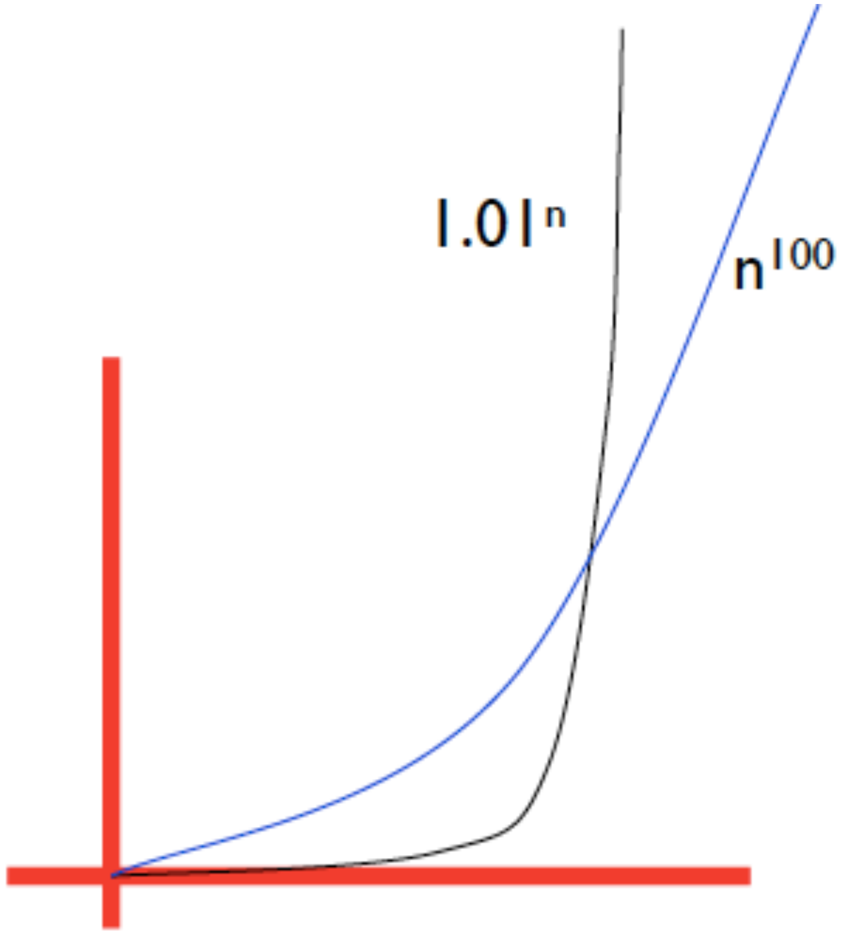
Logarithms vs. polynomial

- Logarithms grow slower than every polynomial
  - For every d > 0 (no matter how small d)
  - $\log n \in O(n^d)$.



Polynomial vs. Exponential

- Exponentials grow faster than every polynomial
  - For every $r > 1$ (no matter how small/close to 1)
  - and every $d > 0$ (no matter how big)
  - $n^d \in O(r^n)$.



Some examples of running times

- O(log n): binary search on sorted list
- O(n): find max element in a list
- O(n log n): sort a set of elements
- $O(n^2)$: find the pair of points that is closest to each other
- $O(n^3)$: Given n sets $S_1, \ldots, S_n$, each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?
- $O(n^k)$: Given a graph of n nodes, are there k nodes such that no two are joined by an edge?
- $O(2^n)$: Enumerate all subsets of a set of n elements
- O(n!): given a set of cities and distances, find a shortest route that visits each city exactly once and returns to the start city? (naive approach)

REPRESENTATIVE PROBLEMS

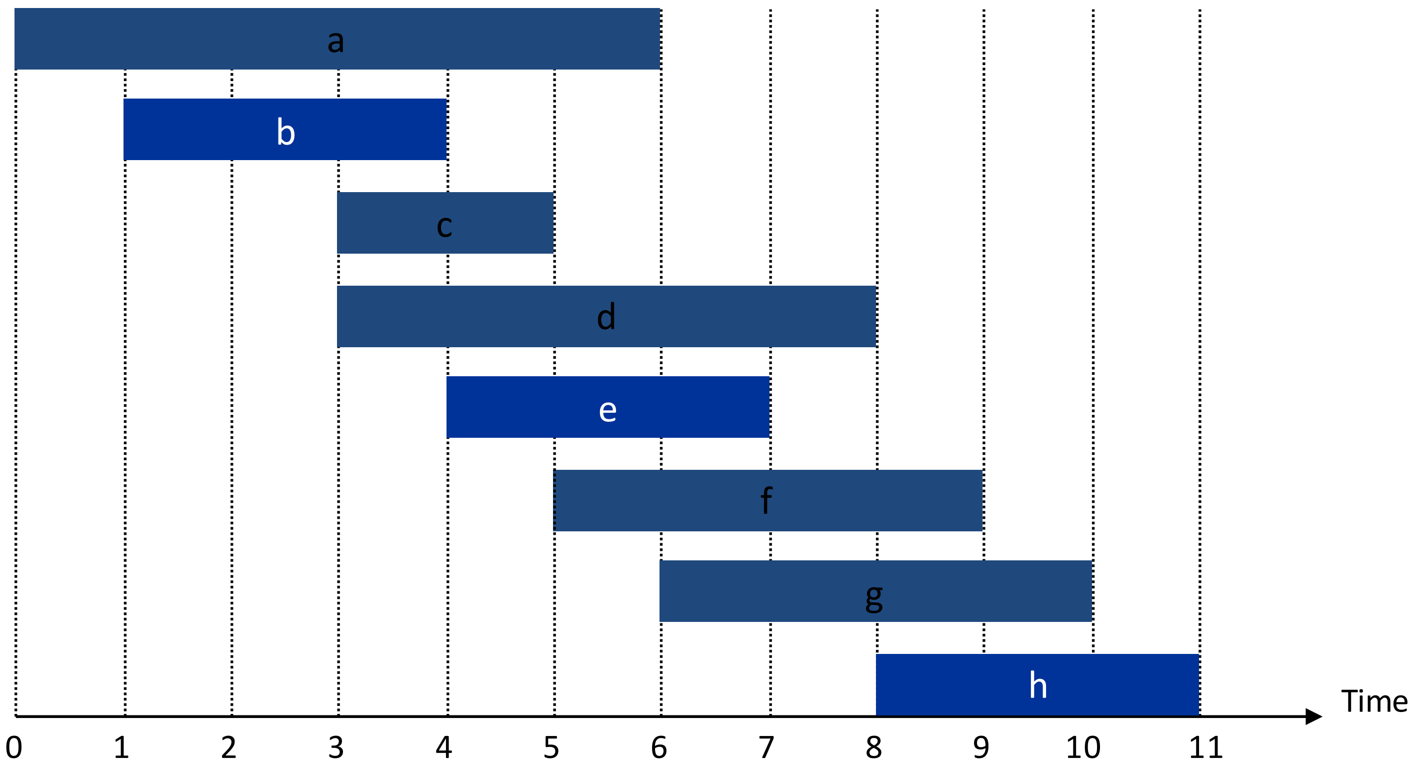
The kind of problems

- Design an algorithm such that
- Given any valid input
  - e.g. a graph, a set of strings from an alphabet, a set of jobs
- Find an output that:
- Satisfies a set of requirements (feasible)
  - e.g. a set of vertices that are not neighbors in the graph
  - e.g. a set of jobs that do not overlap
- And has the best possible quality based on a specified objective function (optimal)

Interval Scheduling

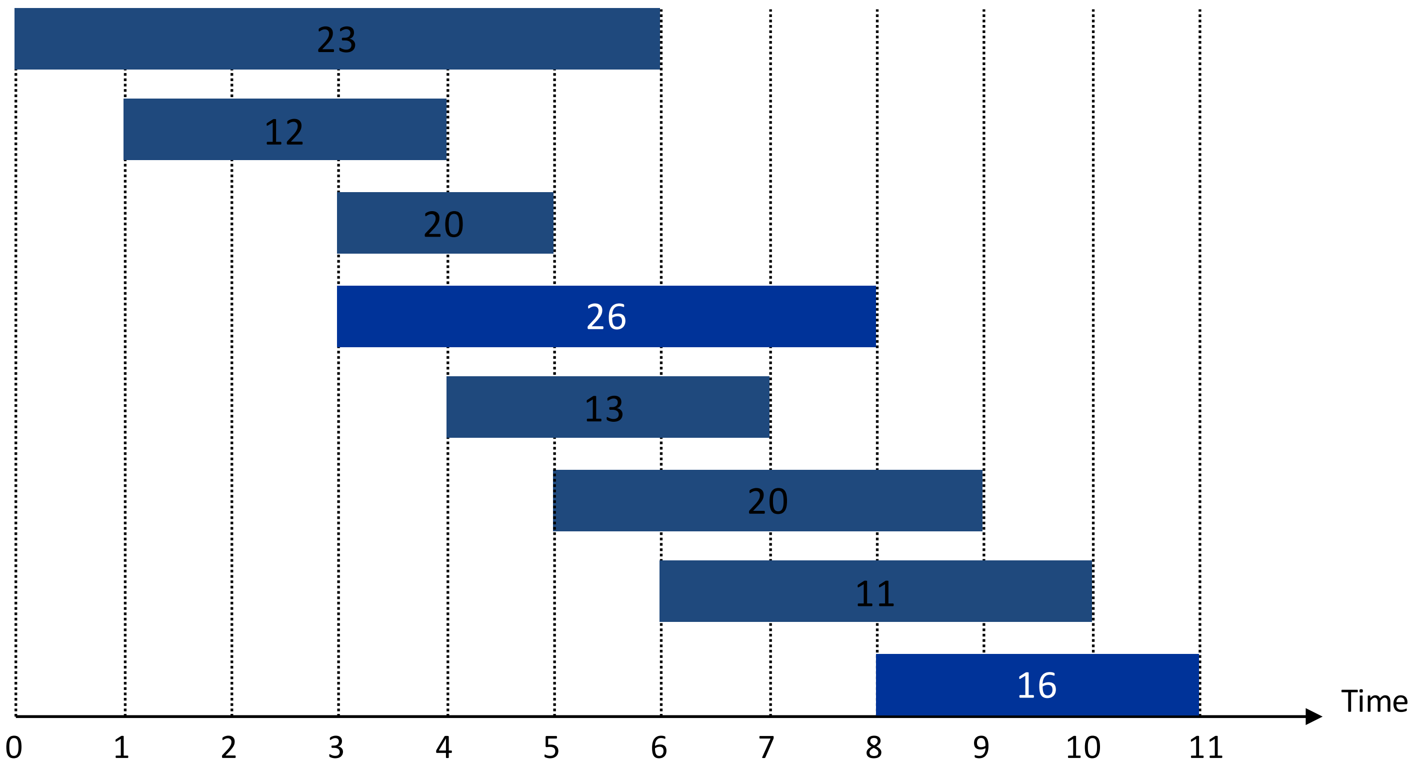
- Input. Set of jobs with start times and finish times.
- Goal. Find maximum cardinality subset of mutually compatible jobs.

jobs don't overlap



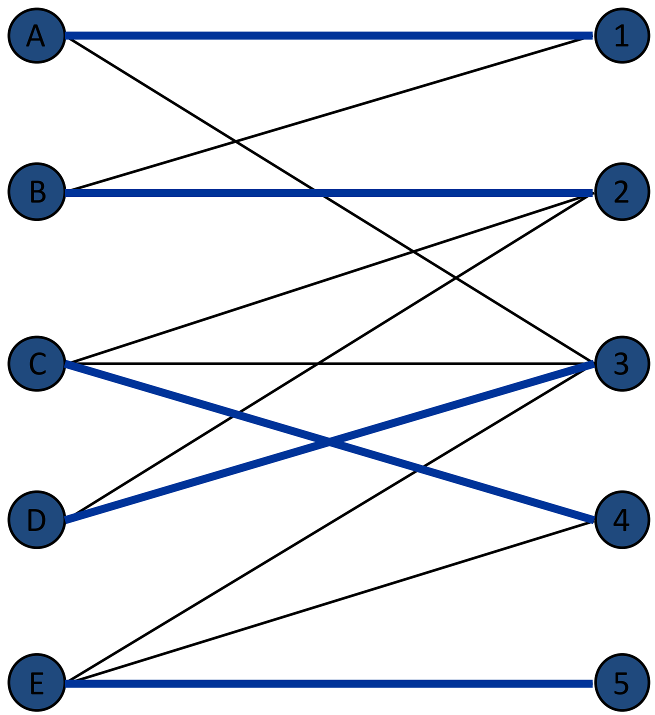
Weighted Interval Scheduling

- Input. Set of jobs with start times, finish times, and weights.
- Goal. Find maximum weight subset of mutually compatible jobs.



Bipartite Matching

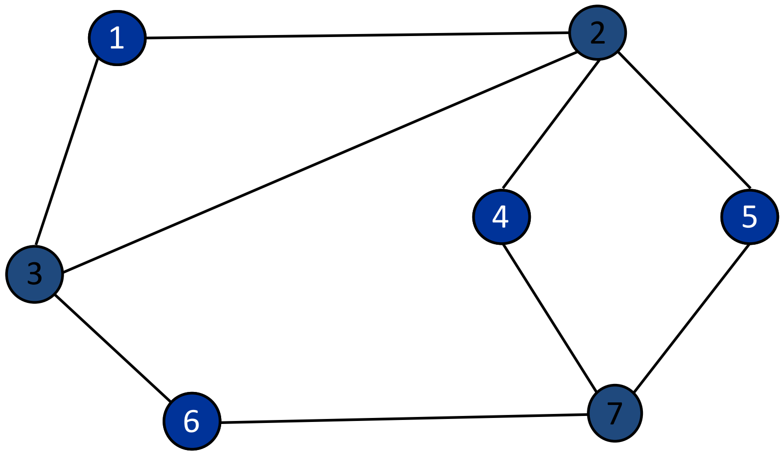
- Input. Bipartite graph.
- Goal. Find maximum number of edges that don't share a node



Independent Set

- Input. Graph.
- Goal. Find maximum cardinality independent set.

subset of nodes such that no two are joined by an edge



Representative Problems

- Variations on a theme: independent set

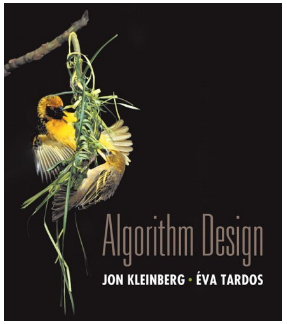
- Interval scheduling: $O(n \log n)$ greedy algorithm
- Weighted interval scheduling: $O(n \log n)$ dynamic programming algorithm
- Bipartite matching: $O(n^k)$ max-flow based algorithm
- Independent set: NP-complete, $O(n^2 2^n)$ brute-force

Greedy Algorithms

Greedy-choice property: we can assemble a globally optimal solution by making locally optimal (greedy) choices

i.e. we make the choice that looks best *given* the current partial solution

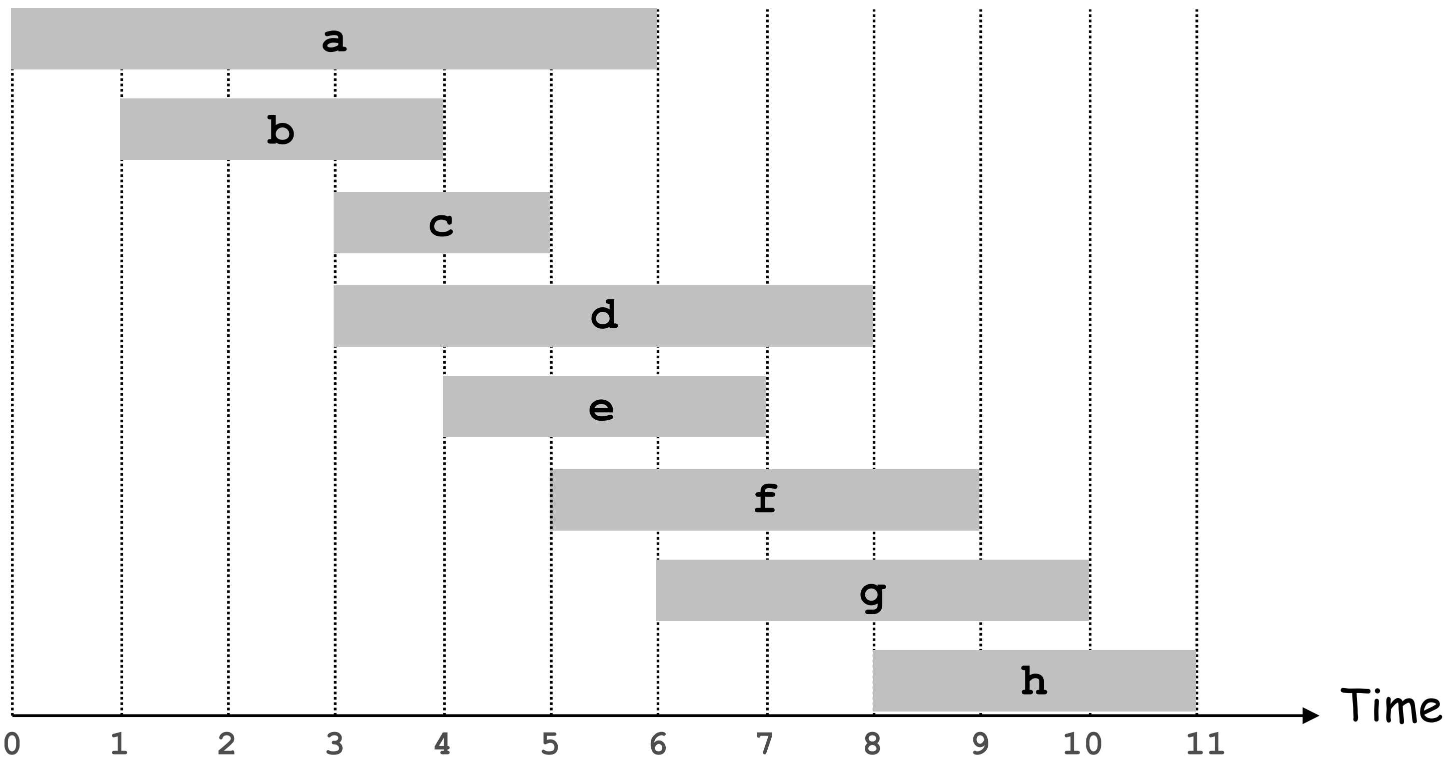
Interval Scheduling [KT 4]



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: Greedy Algorithms

Greedy template. Consider jobs in some natural order.
Take each job provided it's compatible with the ones already taken.

- [Earliest start time] Consider jobs in ascending order of s_j.

counterexample for earliest start time

- [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.

counterexample for shortest interval

- [Fewest conflicts] For each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

counterexample for fewest conflicts

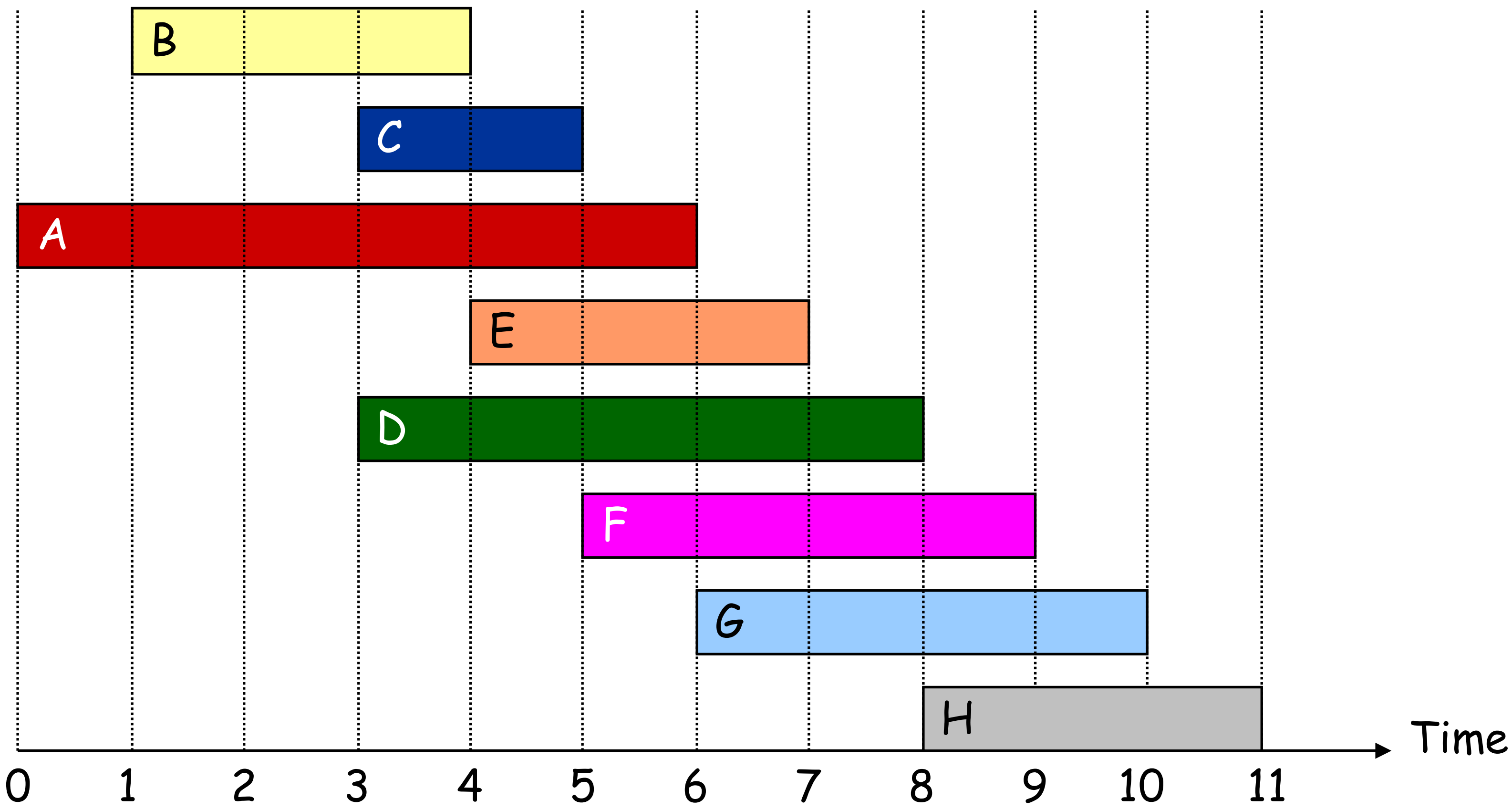
- [Earliest finish time] Consider jobs in ascending order of f_j.

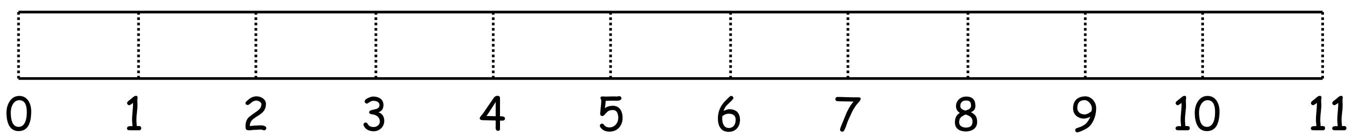
Interval Scheduling: Greedy Algorithm

Greedy algorithm. Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken. (natural order = finish time)

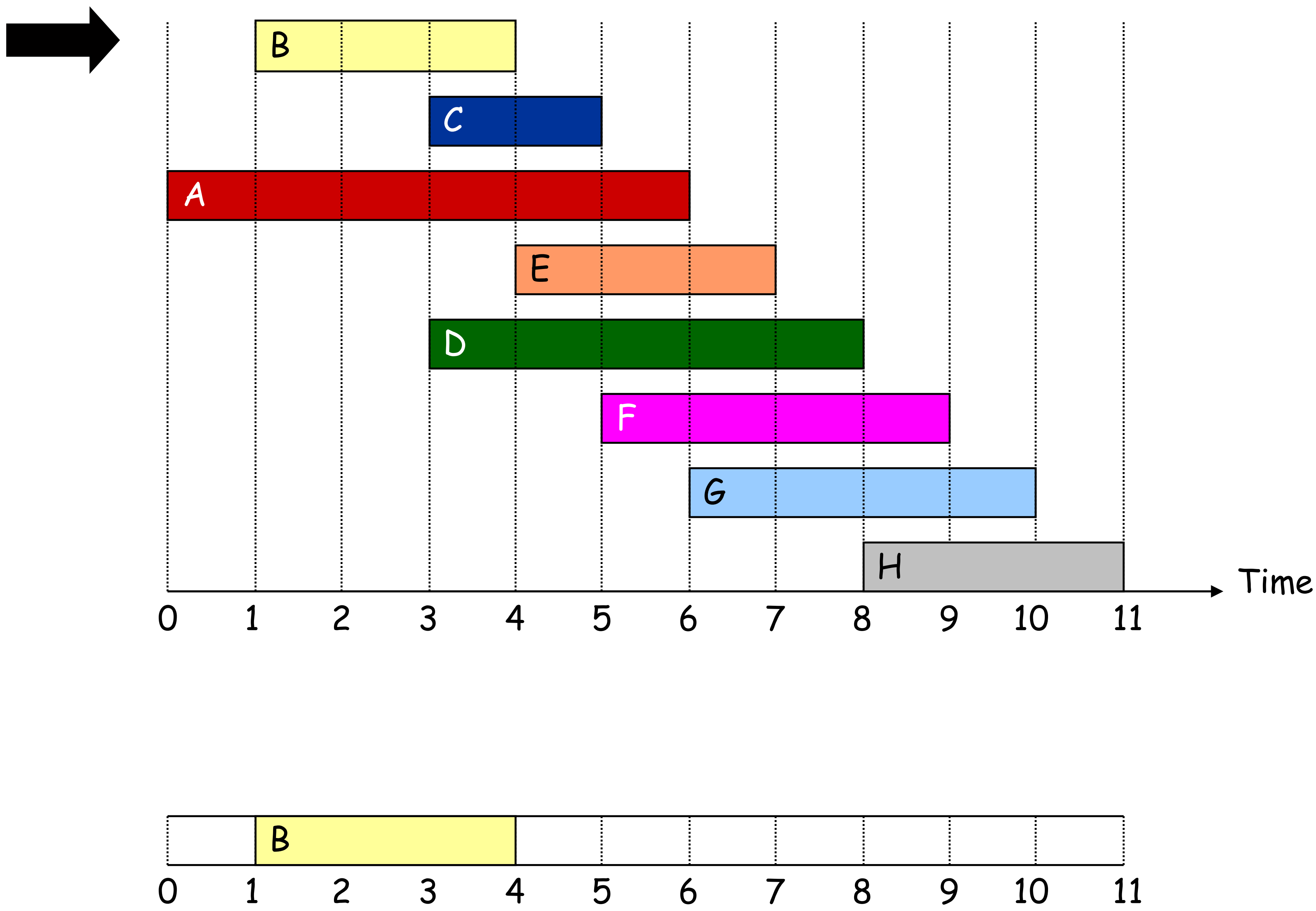
```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

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

Interval Scheduling

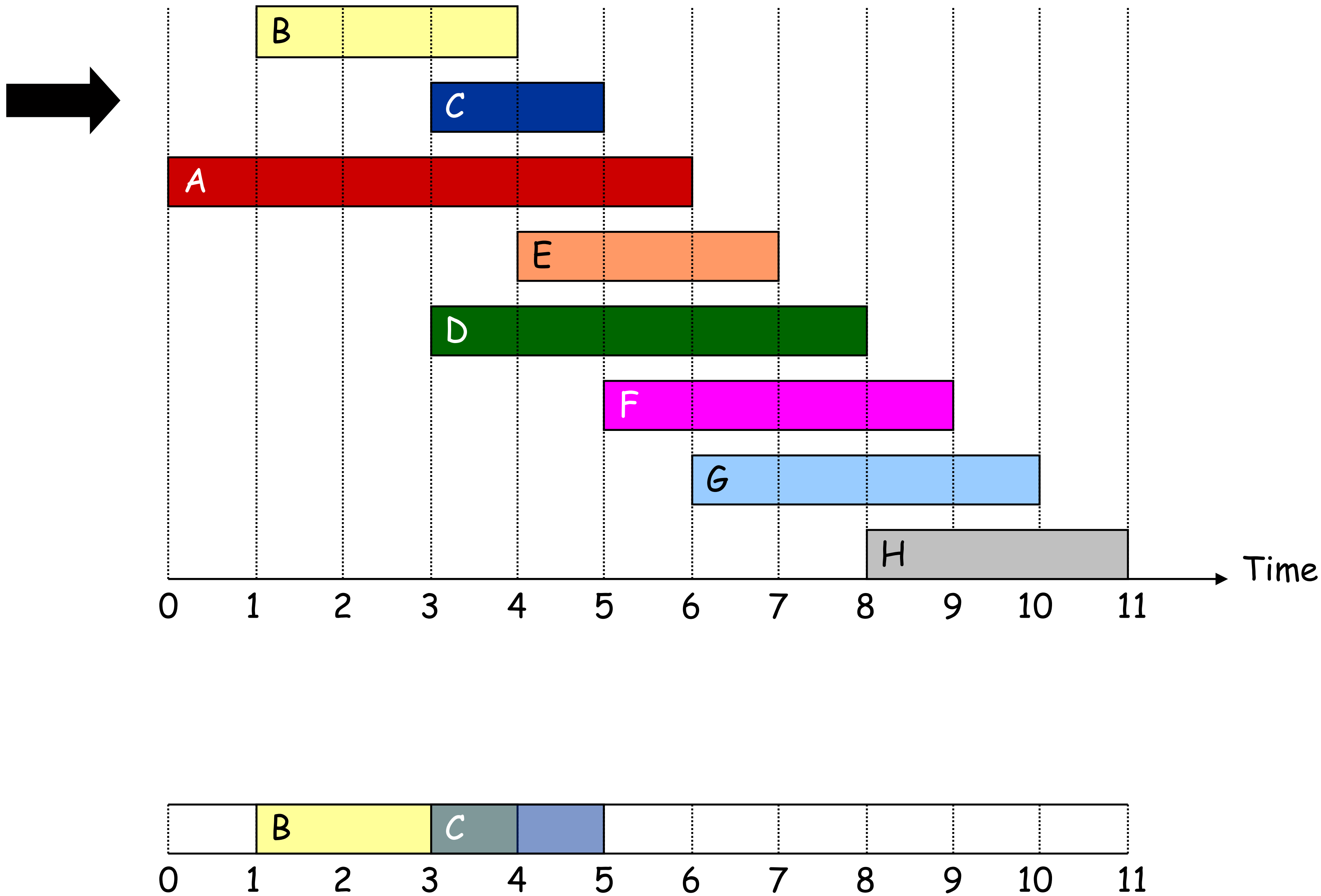




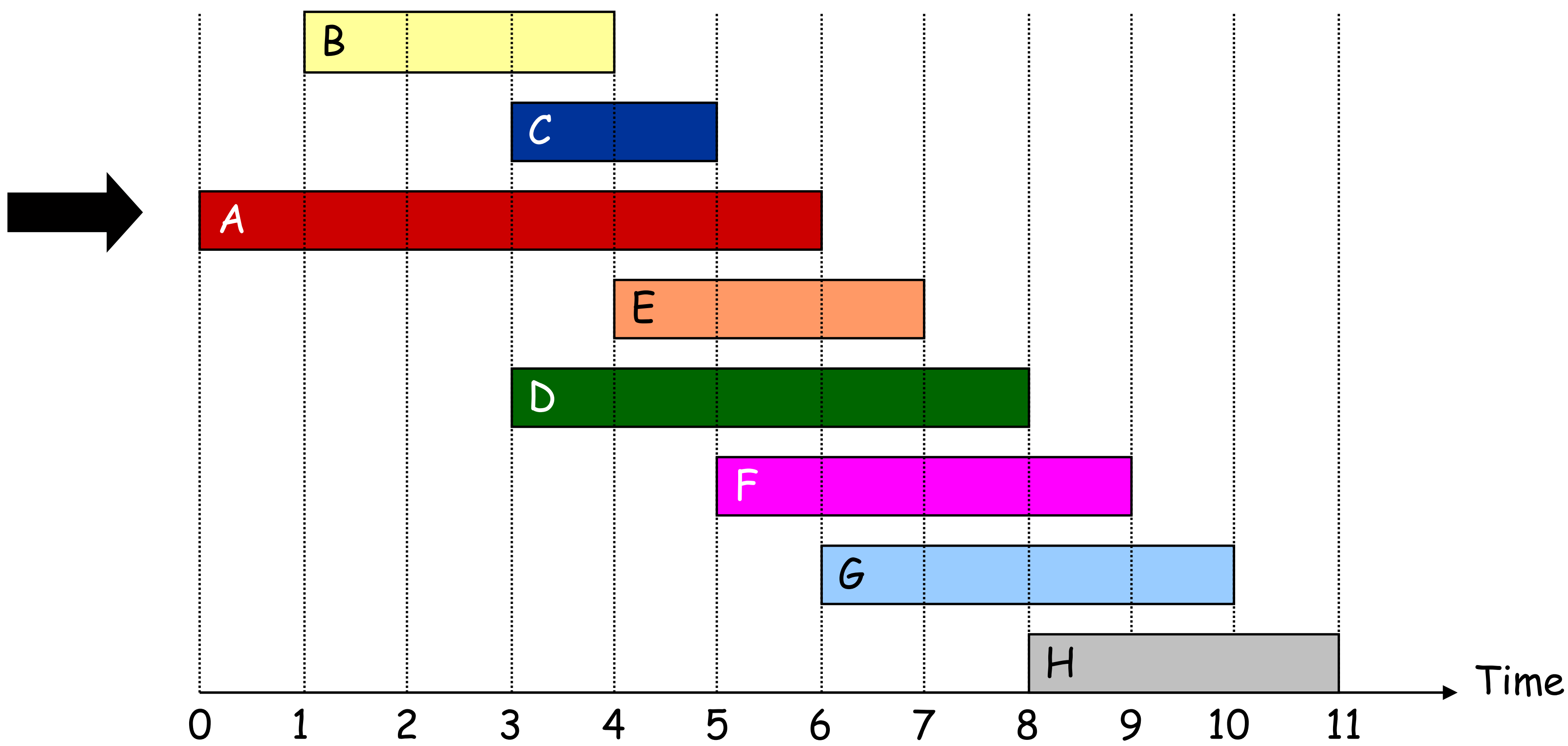
Interval Scheduling

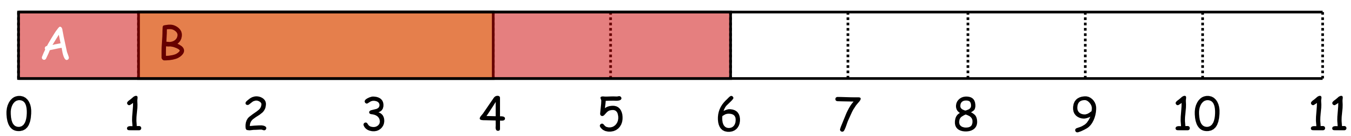


Interval Scheduling

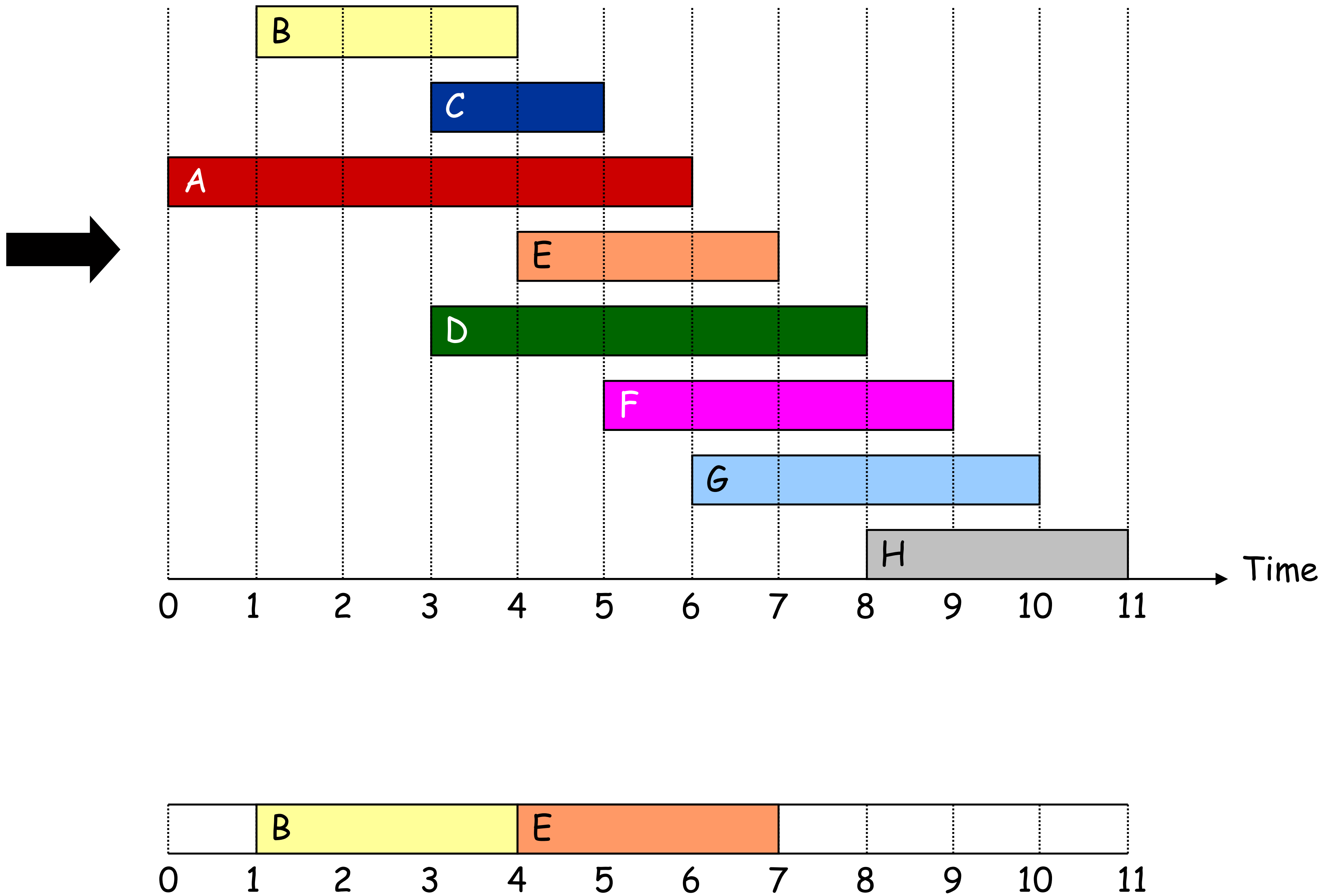


Interval Scheduling

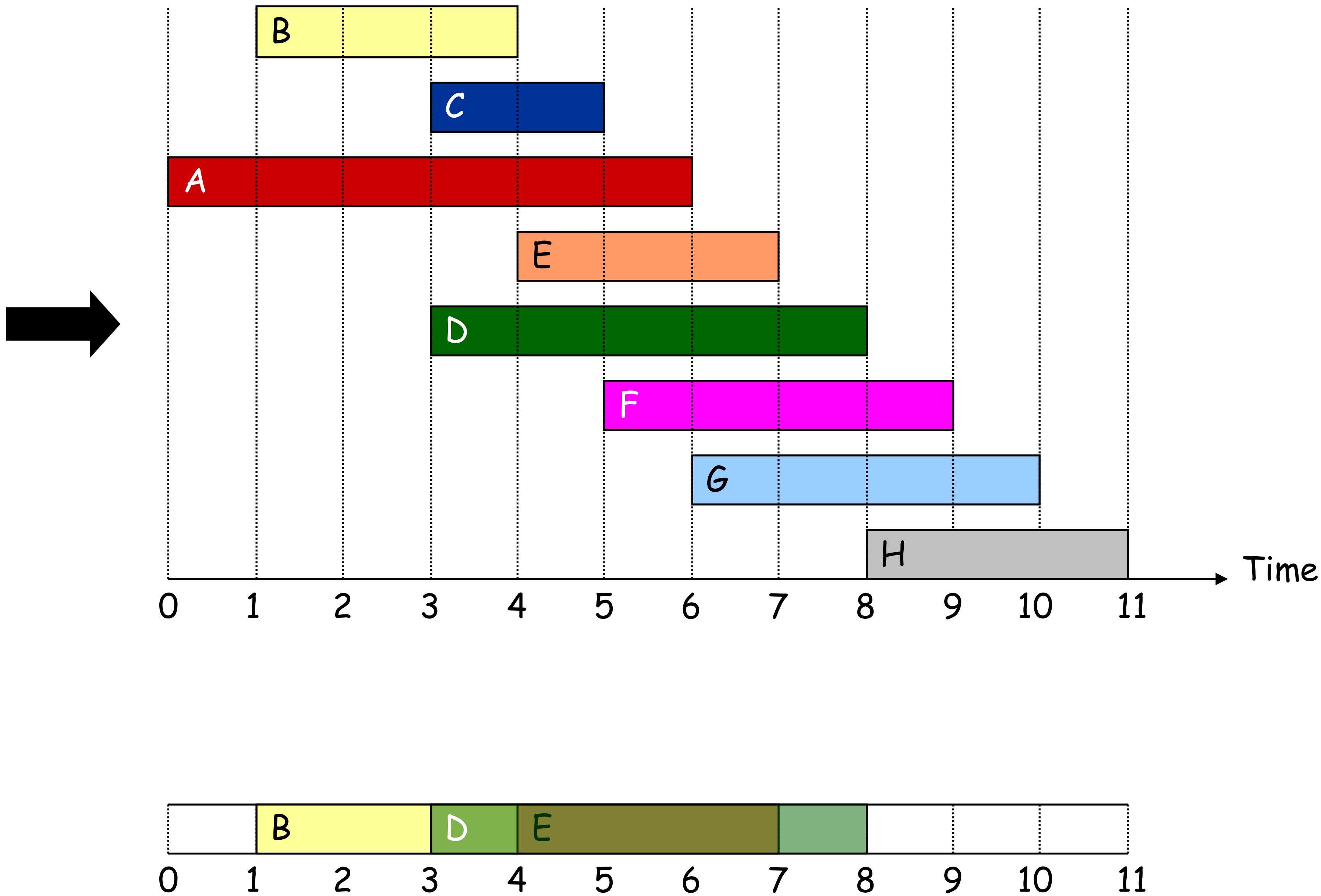




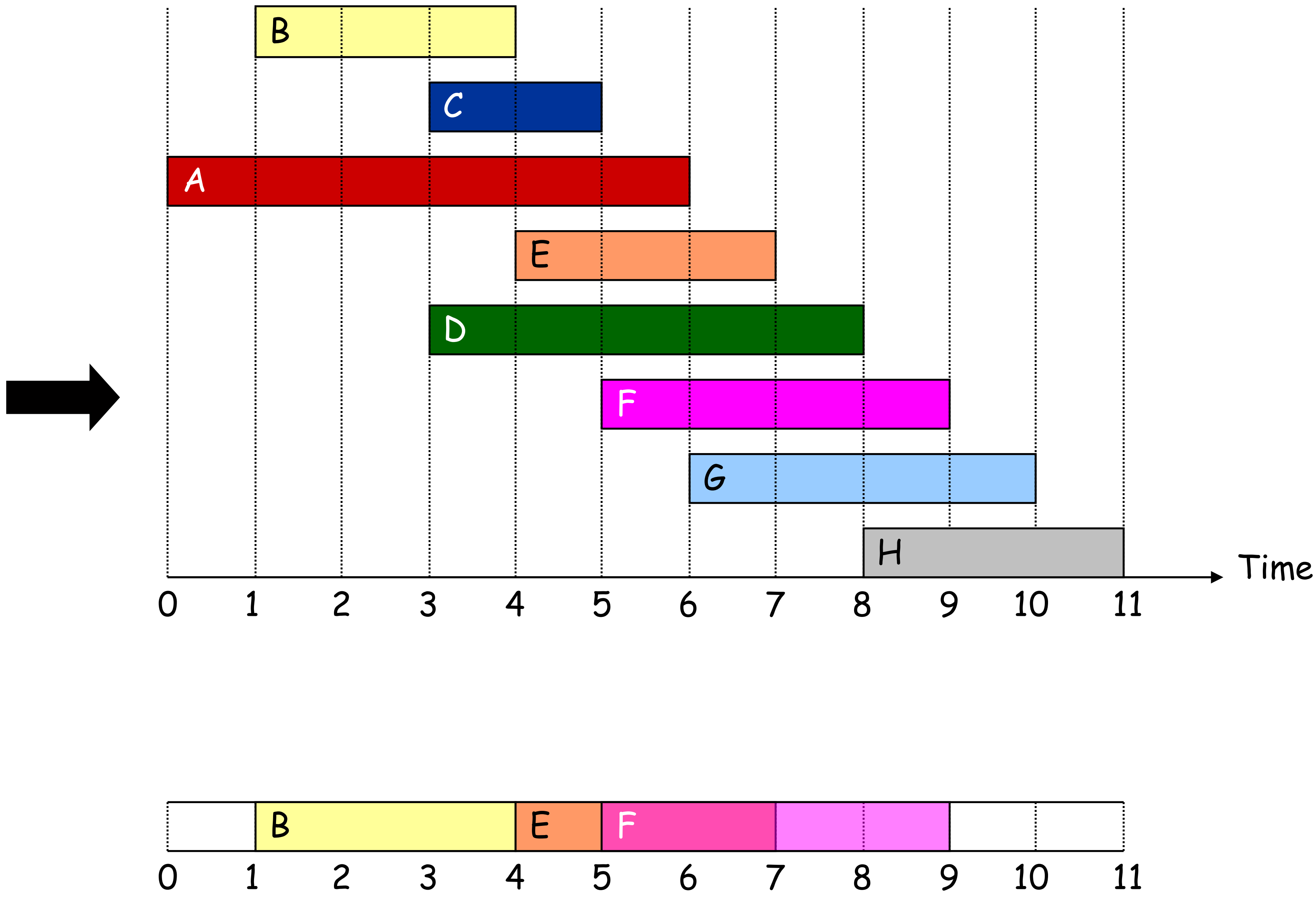
Interval Scheduling



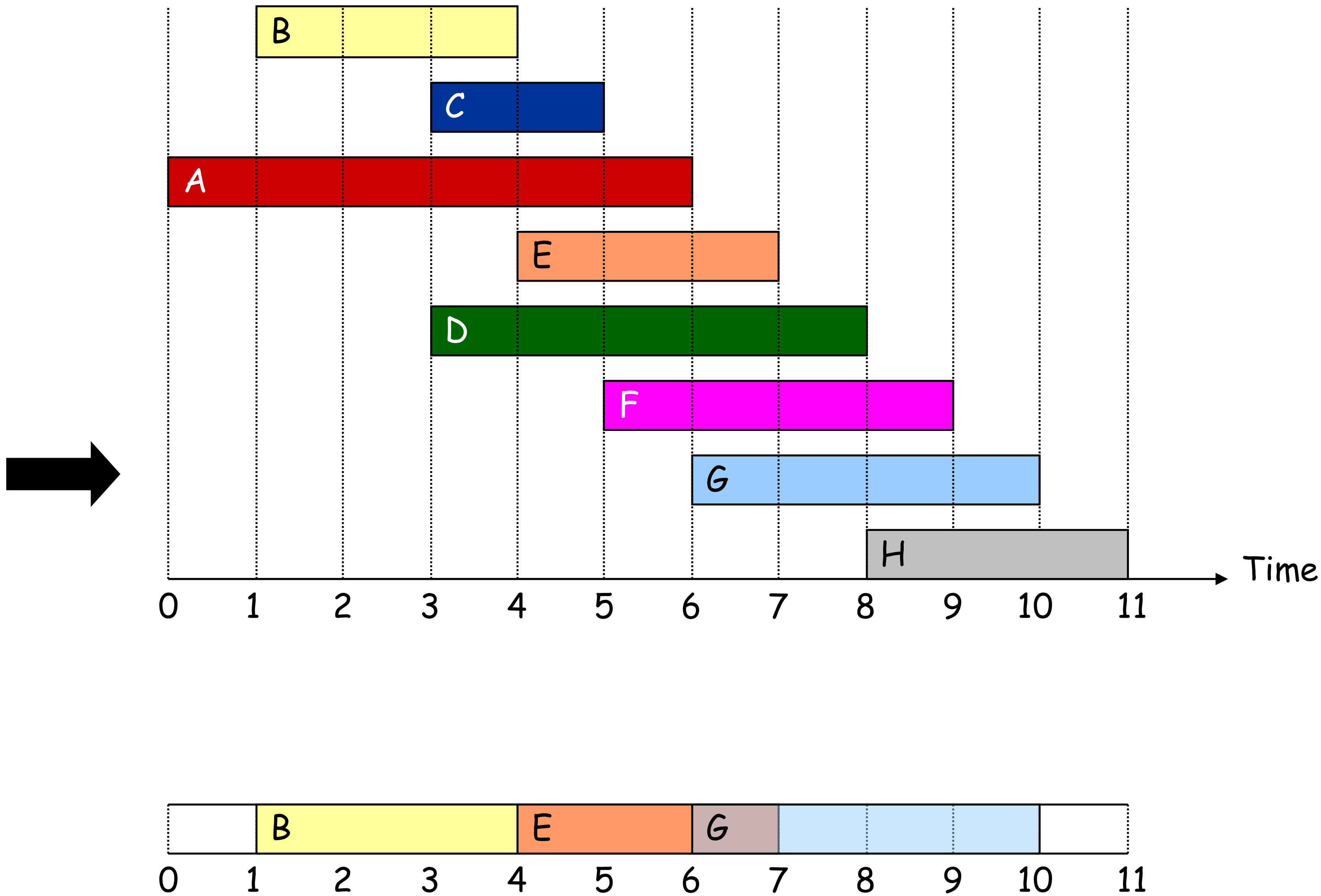
Interval Scheduling



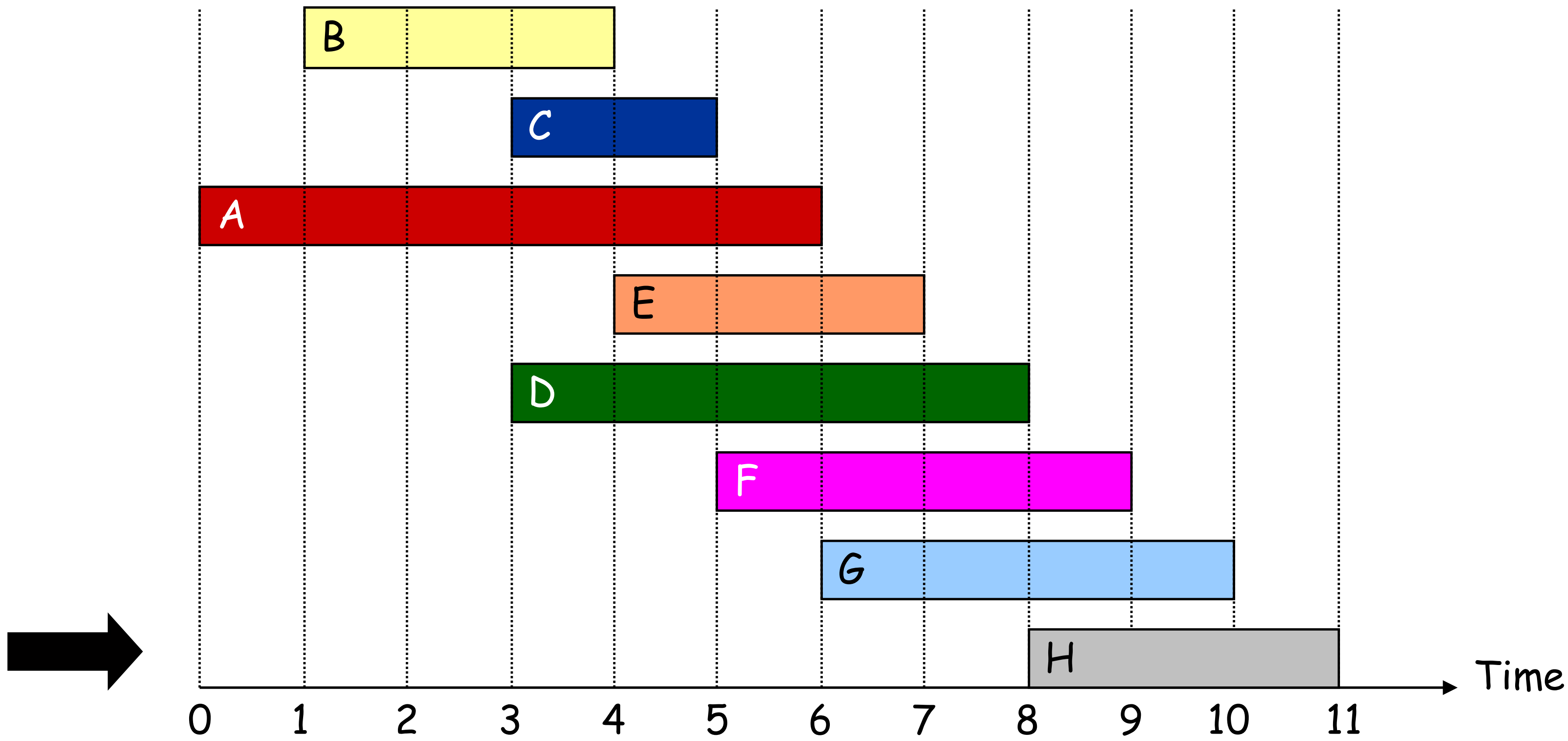
Interval Scheduling

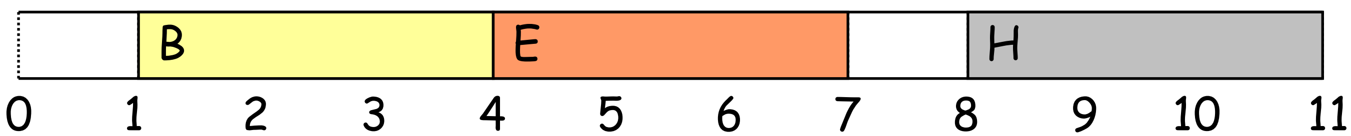


Interval Scheduling



Interval Scheduling





Interval Scheduling: Greedy Algorithm

Greedy algorithm. Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.

O(n log n)

Naïve $O(n^2)$
Naïve O(n)

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

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

Running time. $O(n^2)$.

Why is this optimal?
Come to next class!

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.

O(n log n)

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

   set of jobs selected
A ← φ
fj* = 0
for j = 1 to n {
   if (job j compatible with A: sj ≥ fj*)
      A ← A ∪ {j}
      fj* = fj
}
return A
```

O(n)

Running time. O(n log n).

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