CS340 - Analysis of Algorithms

Dynamic Programming

Announcements:

Hw5 Due Monday November 3rd

Lab 5 due tomorrow

Divide and Conquer Quiz - November 6th in Lab

CS Major info session tomorrow 4pm

Project Checkpoint 2 due tomorrow:

- Actually run your program
- Plot runtime with increasing sizes of problems (use script to generate different inputs increasing your dominant variable)
- If the curve is not obvious, run regression

Divide and Conquer Review

- Merge Sort:
  - Brute force $O(n^2)$
  - Divide and conquer: $O(n\log n)$
- Inversion Counting:
  - Brute force: $O(n^2)$
  - Divide and conquer: $O(n\log n)$
- Closest Pair:
  - Brute force: $O(n^2)$
  - Divide and conquer: $O(n\log n)$

Divide and Conquer is effective at reducing polynomial run time down to a faster polynomial

Not strong enough to reduce an *exponential* brute force down to polynomial time

Dynamic Programming

Dynamic Programming

- Implicitly explore the space of all possible solutions
- Decompose into series of subproblems
- Build up the correct solutions to larger and larger subproblems
- Operates dangerously close to the edge of brute force
- Works through the exponentially large set of possible solutions, but does not examine them all explicitly

Dynamic Programming

- Smart recursion - *without repetition*
- Stores the solutions of intermediate subproblems, usually in tables (arrays)
- Optimization problems that can be solved by a greedy algorithm are VERY rare
- Your first instinct should be DP, not greedy

DP Design

Break down the problem by **expressing the optimal solution in terms of optimal solutions to sub-parts**

- Subproblems may overlap
- The number of subproblems must be reasonably small

To do this, determine the recursive formulation

- First describe the precise function you want to evaluate in English
- Then, give a formal recursive definition of the function

Needs an evaluation order and a measure of “optimal”

Dynamic Programming

Find a (small) choice whose correct answer would reduce the problem size

For each possible answer, temporarily adopt that choice and recurse

Don’t be clever with choices, try them all!

Weighted Interval Scheduling

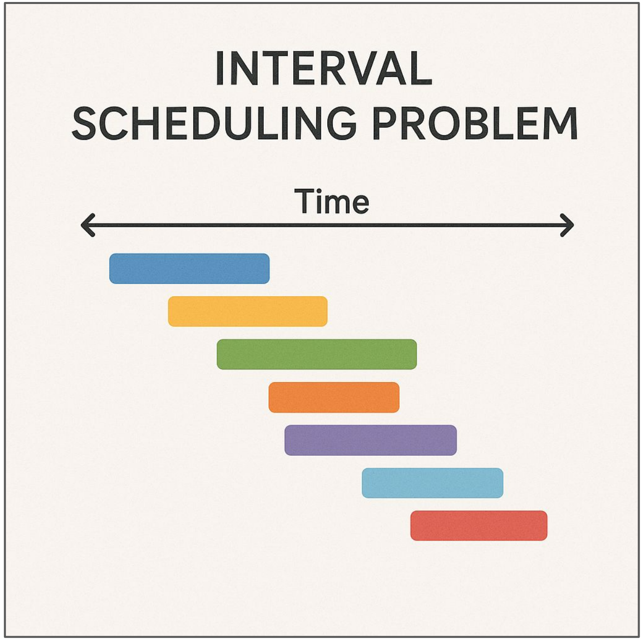
Interval Scheduling

You have a resource (lecture room, supercomputer, pool, hockey rink, electron microscope...) that can be used by at most one person / group at a time

Many requests come in to use that resource for periods of time

A request takes the form: *Can I reserve the resource starting at time **s** until time **f**?*

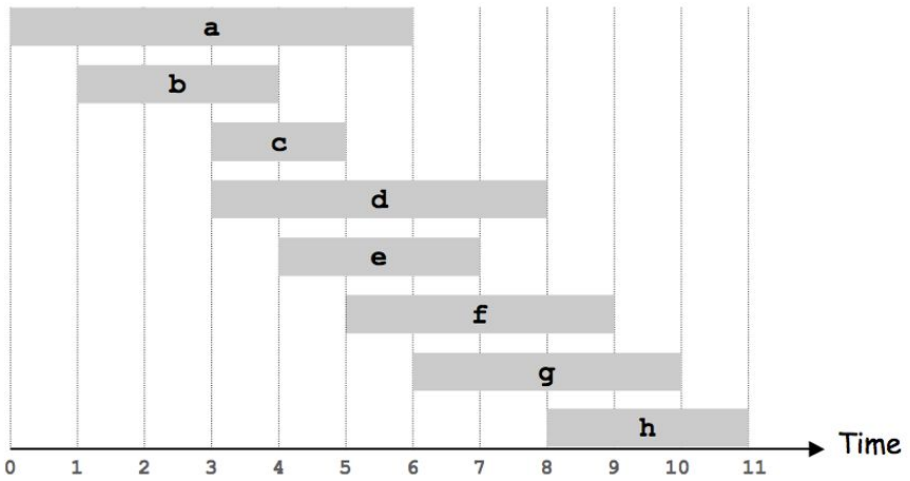
Goal: schedule as many requests as possible



Interval Scheduling

- Given a set R of n activities with start-finish times $[s_i, f_i], 1 \leq i \leq n$, determine a maximum subset of R consisting of compatible requests

What does compatible mean?



Interval Scheduling - compatibility

Two requests i and j are compatible if the requested intervals do not overlap:

Either

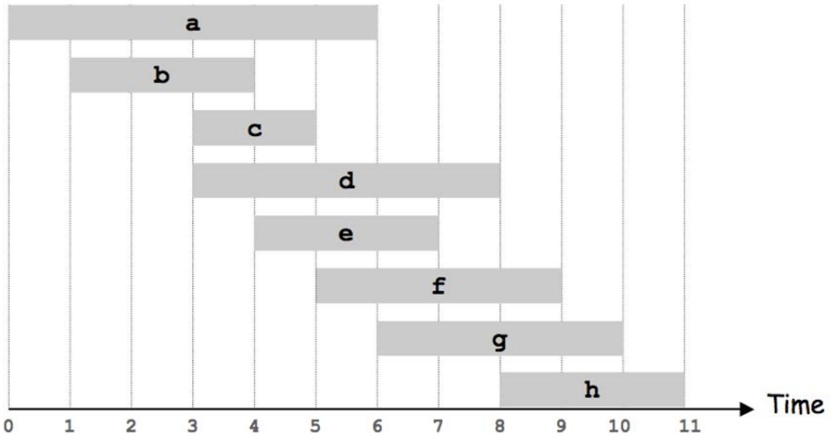
(a) request i is for an earlier time interval than request j ($f_i \leq s_j$) OR
(b) Request i is for a later time than request j ($f_j \leq s_i$)

A subset A of requests is compatible if all pairs of requests i, j $\in$ A, i != j are compatible.

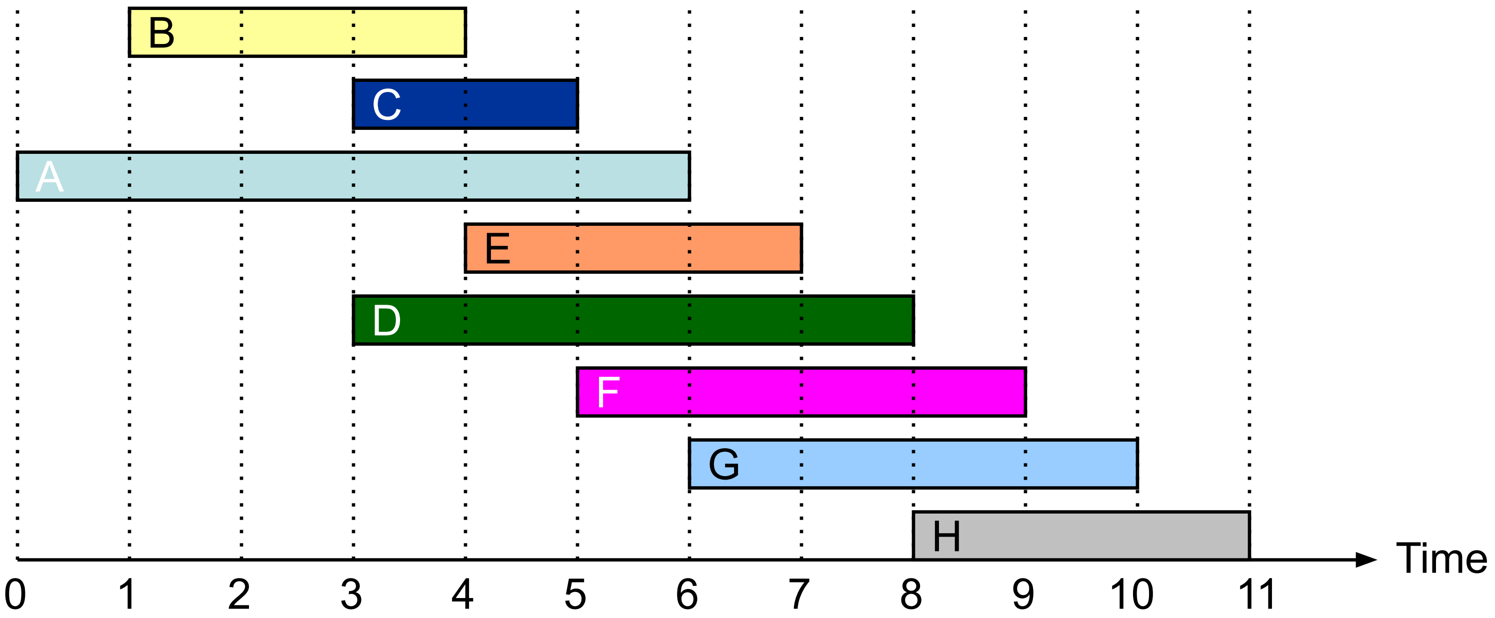
Goal of interval scheduling is to select a compatible subset of requests of **maximum size**

A Greedy Solution

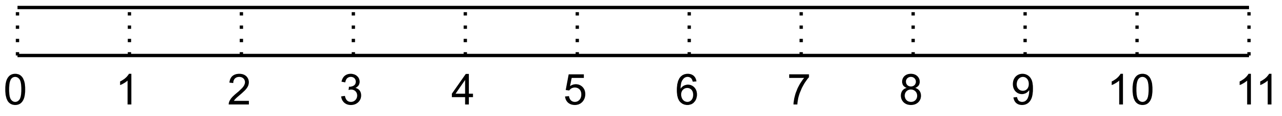
- For each request, use simple criteria to decide if it should be accepted
- Once accepted it can not be rescinded
  - **Greedy algorithms do not backtrack**
- **Select the interval that finishes first**



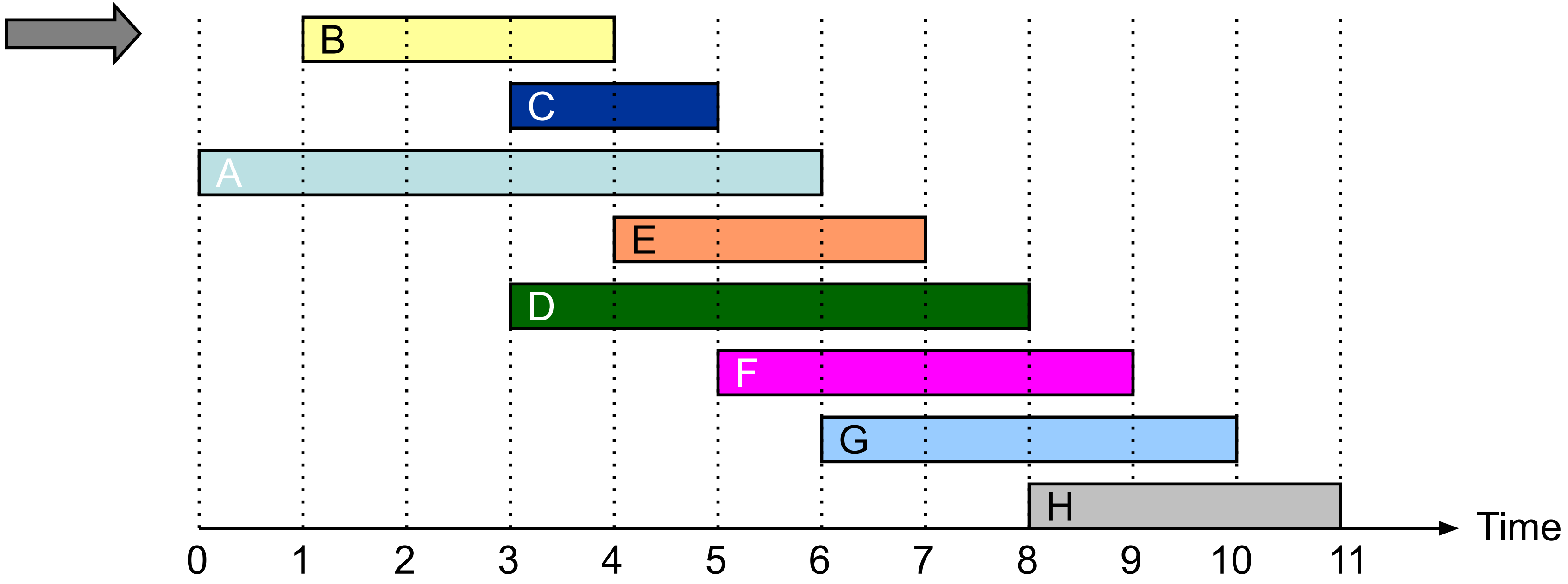
Interval Scheduling: Select Earliest Finish

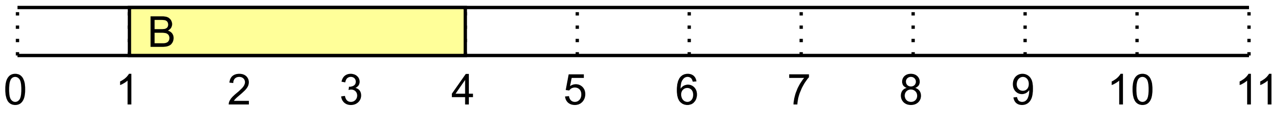


Fill in the schedule:

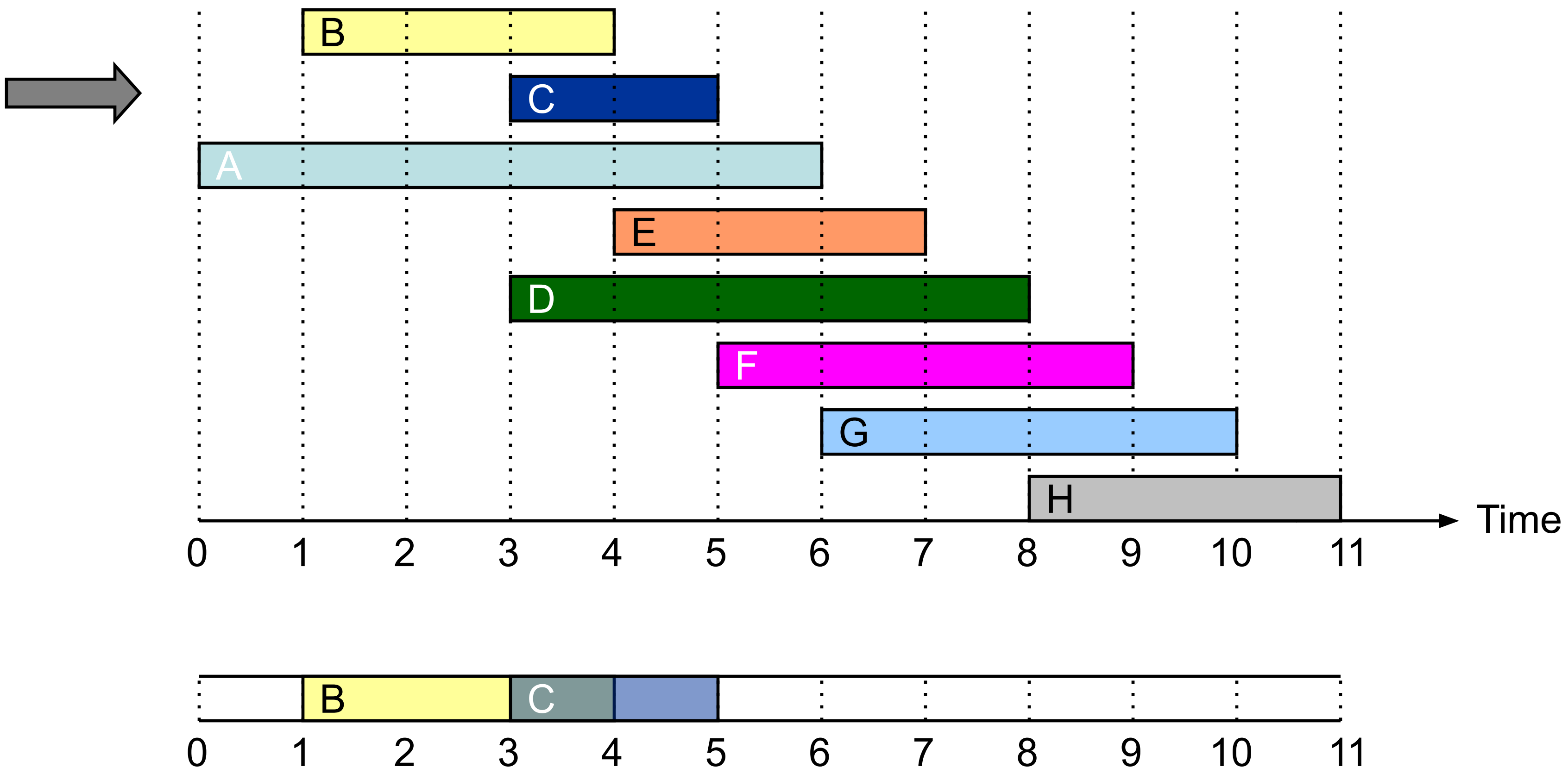


Interval Scheduling: Select Earliest Finish

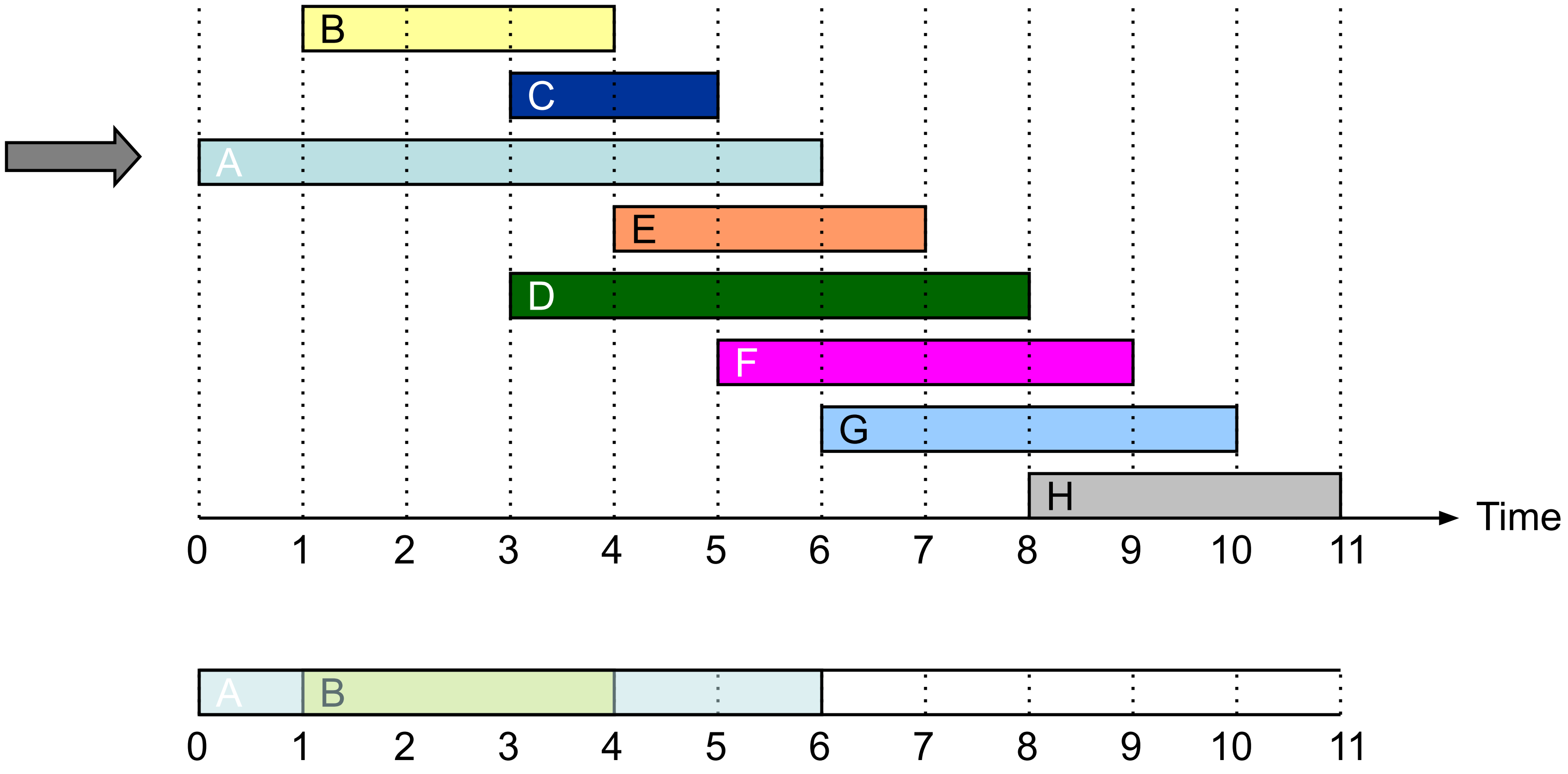




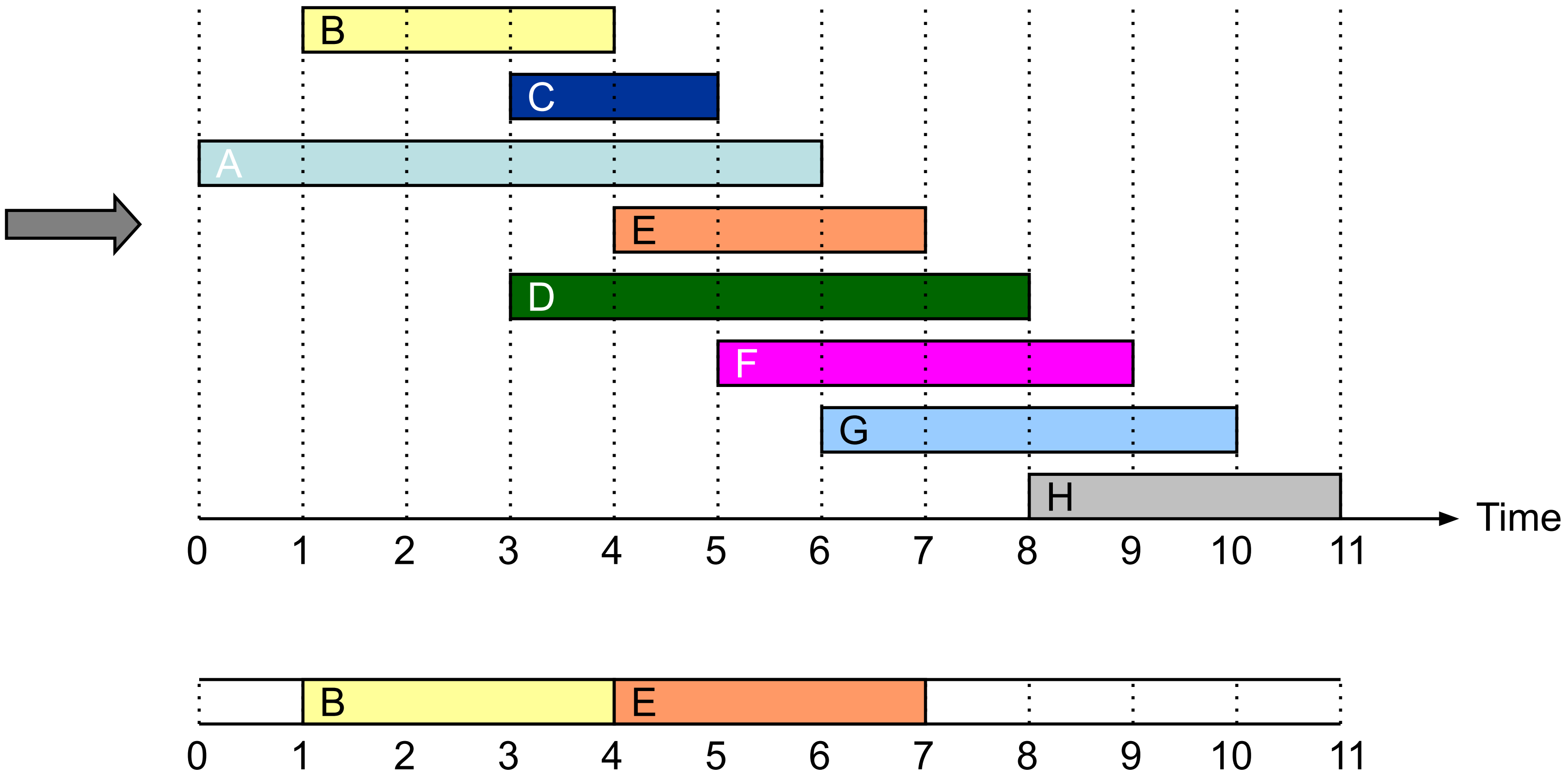
Interval Scheduling: Select Earliest Finish



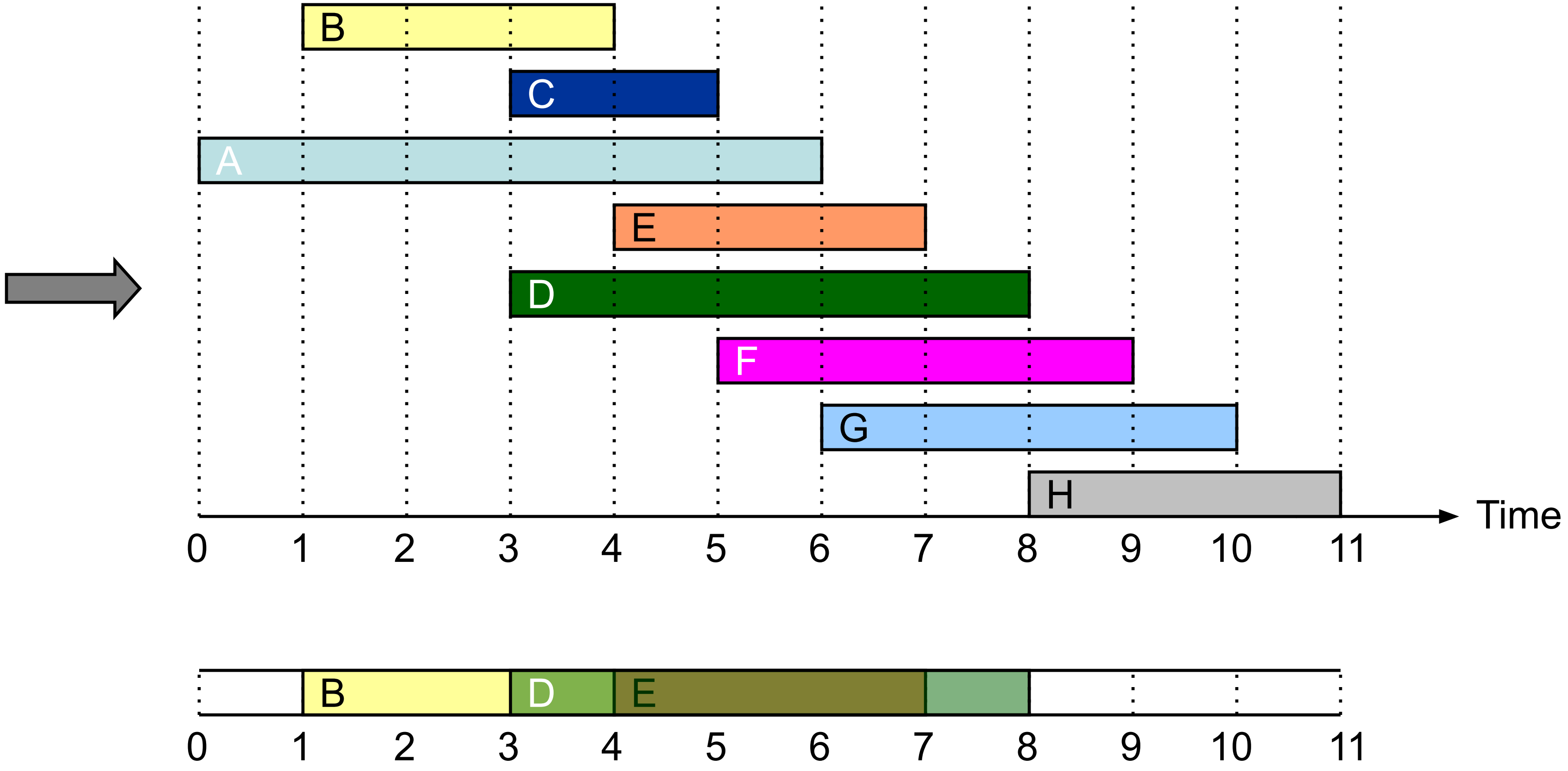
Interval Scheduling: Select Earliest Finish



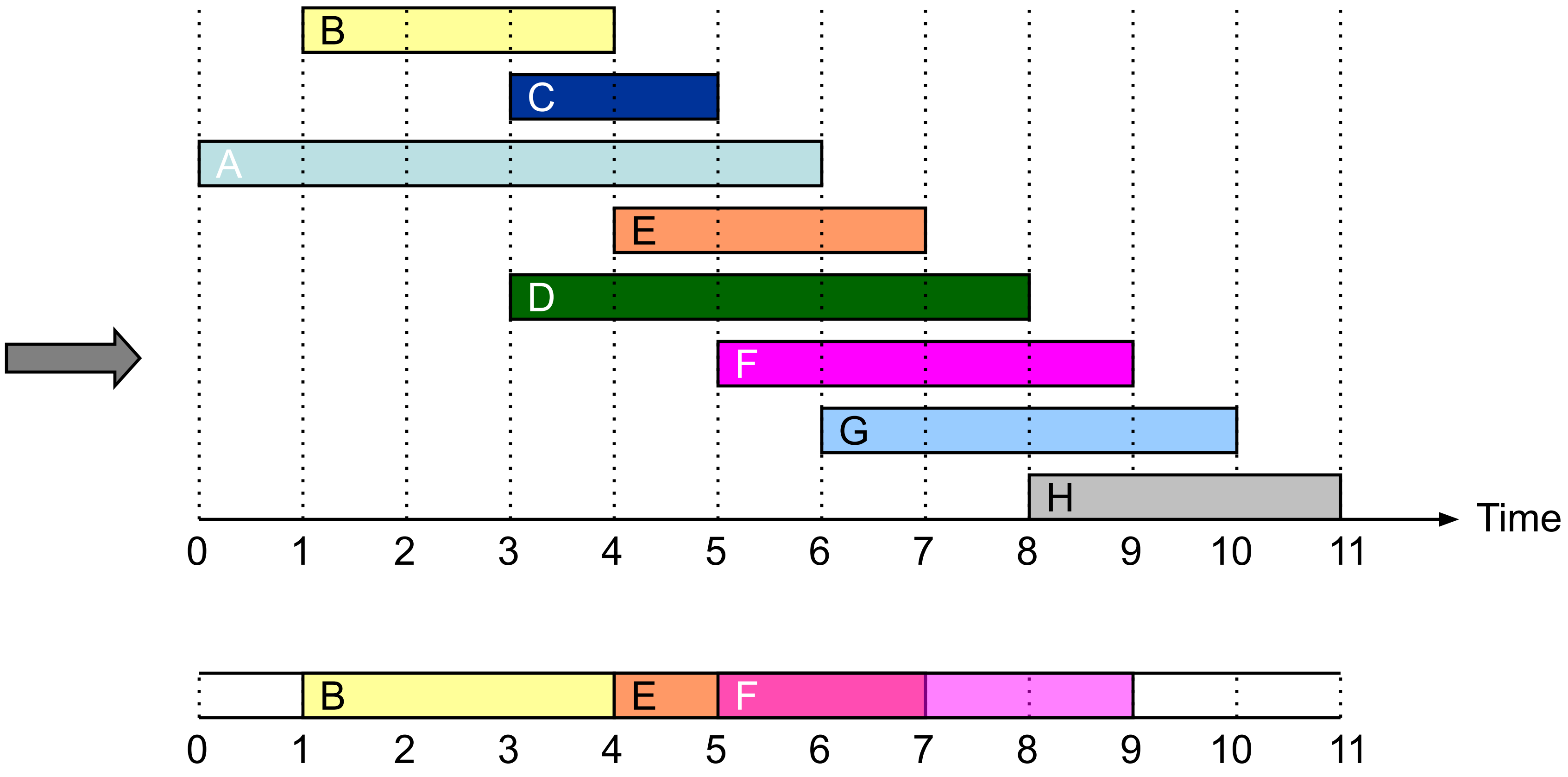
Interval Scheduling: Select Earliest Finish



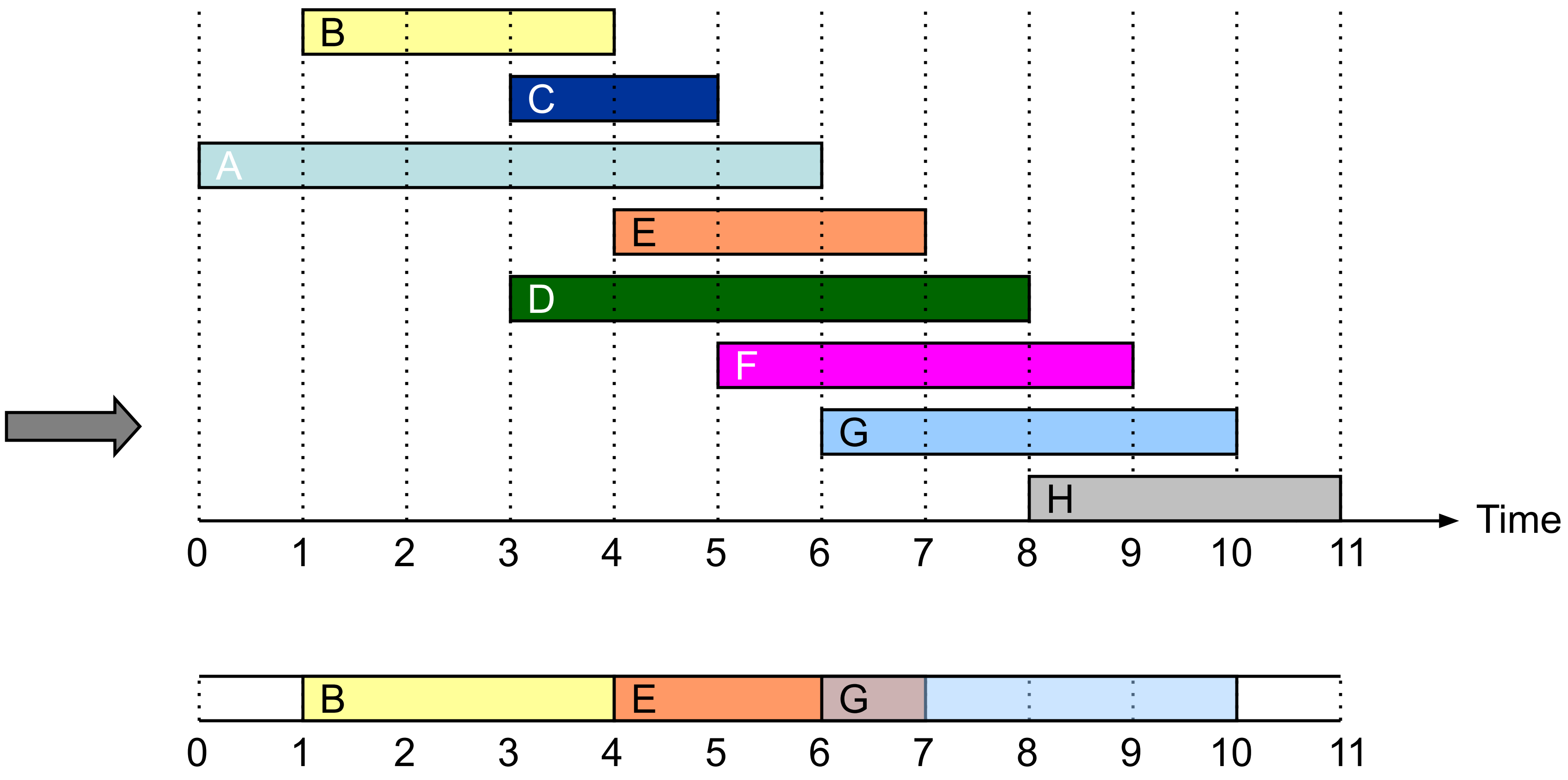
Interval Scheduling: Select Earliest Finish



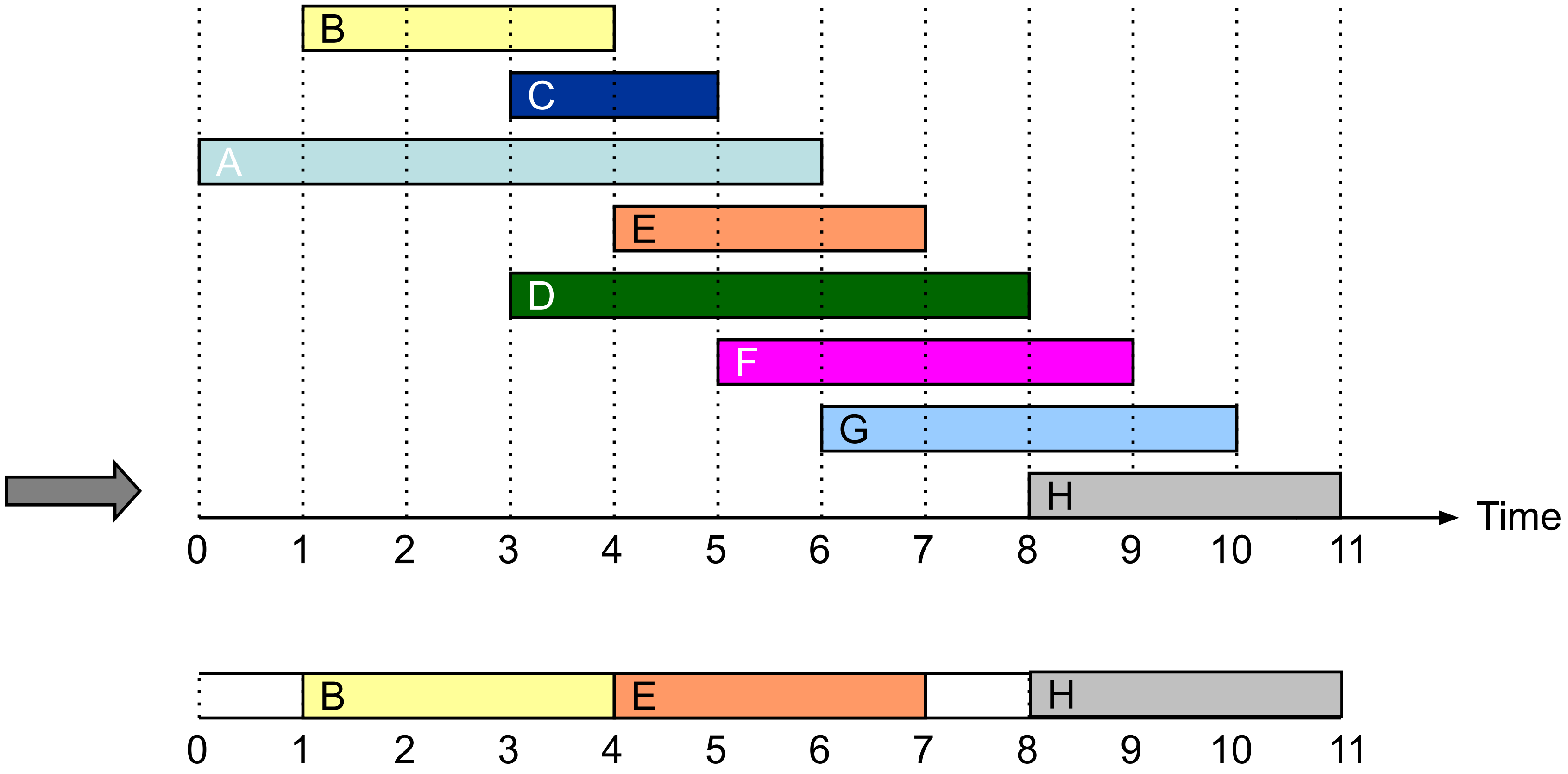
Interval Scheduling: Select Earliest Finish



Interval Scheduling: Select Earliest Finish



Interval Scheduling: Select Earliest Finish



Select Earliest Finish

```
greedySchedule(R) { // R the set of requests

    A = empty; // A the set of scheduled activities
    sort R by finish times
    prevA = null //last picked activity

    for each (r in R) {
        if (r does not conflict with prevA) {
            append r to A;
            prevA = r;
        }
    }
    return A;
}
```

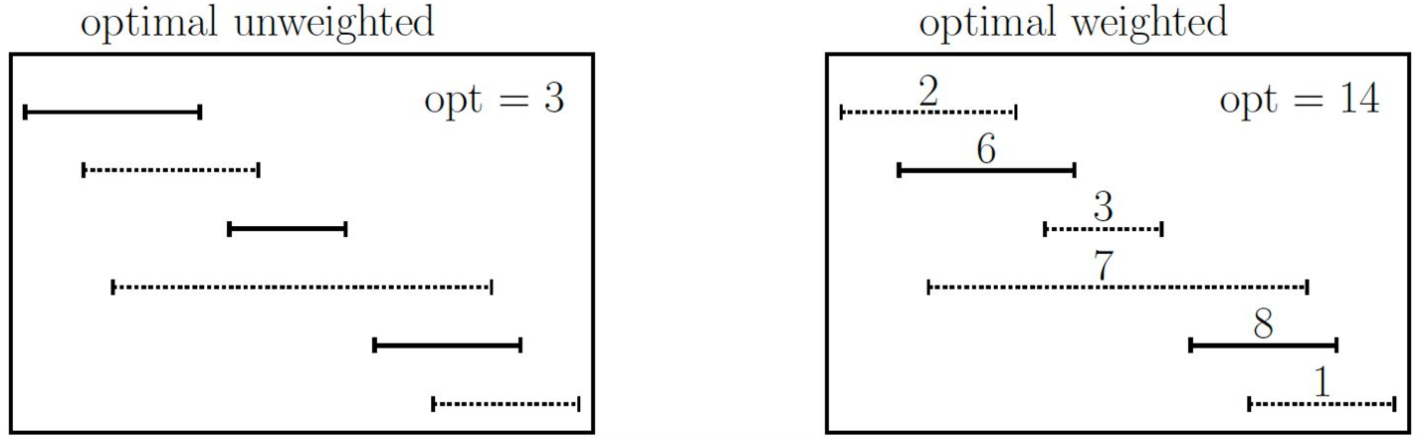
Greedy runtime?

$O(n\log n + n) =$

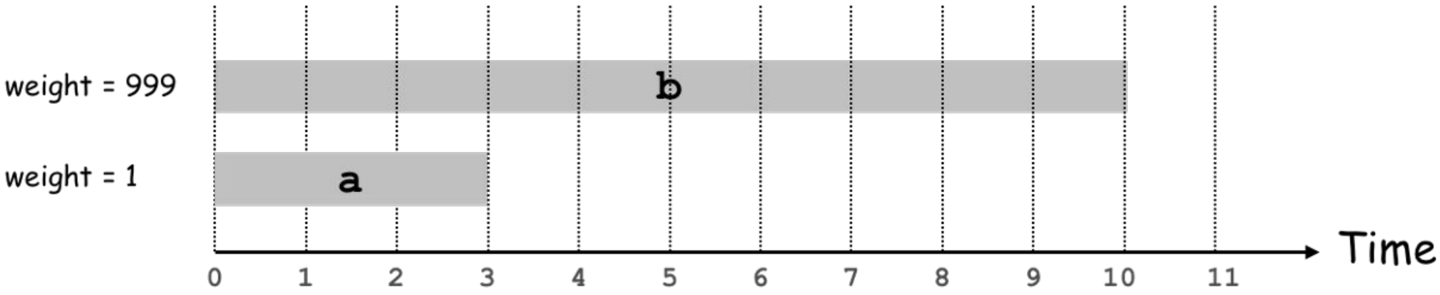
$O(n\log n)$

Weighted Interval Scheduling

- A more general version of interval scheduling
- Each interval also has a weight w_i
- Find a set of compatible intervals that have maximized total weights



Weighted Interval Scheduling

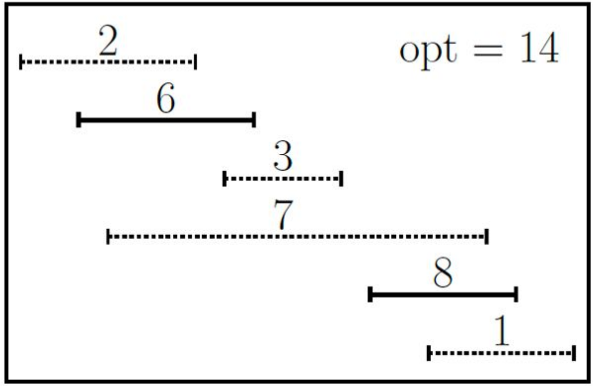


No greedy solution is known!

Weighted Interval Scheduling

Brute force?

$O(2^n)$



Weighted Interval Scheduling

- Given a set R of n activities with start-finish times $[s_i, f_i], 1 \leq i \leq n$

- **Goal**: select a subset of compatible intervals S as to maximize the sum of weights:

$$\sum_{i \in S} w_i$$

Weighted Interval Scheduling

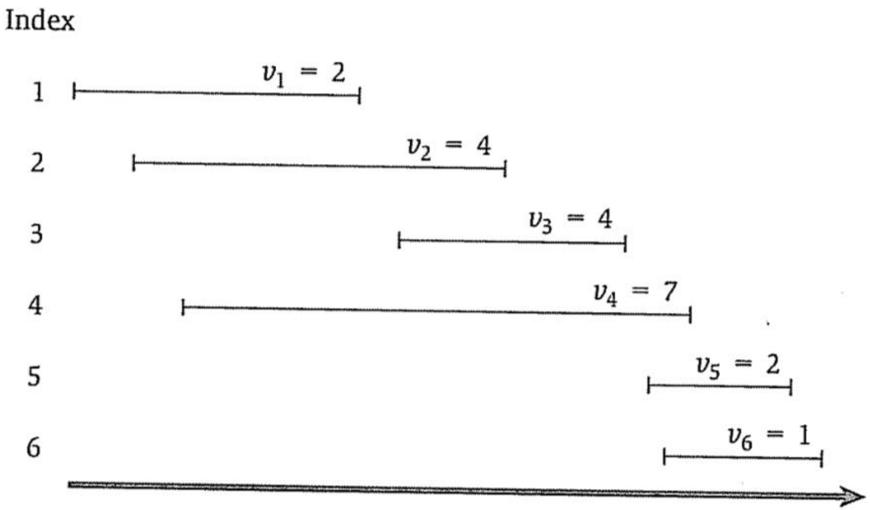
- Suppose requests are given in order of non-decreasing finish time
  - $f_1 \leq ... \leq f_n$

- "*i comes before j*"
  - if i < j in this ordering

- We define a p(j) for an interval j
  - Returns the largest index i < j such that i and j are compatible
  - p(j) = 0 if no such interval exists

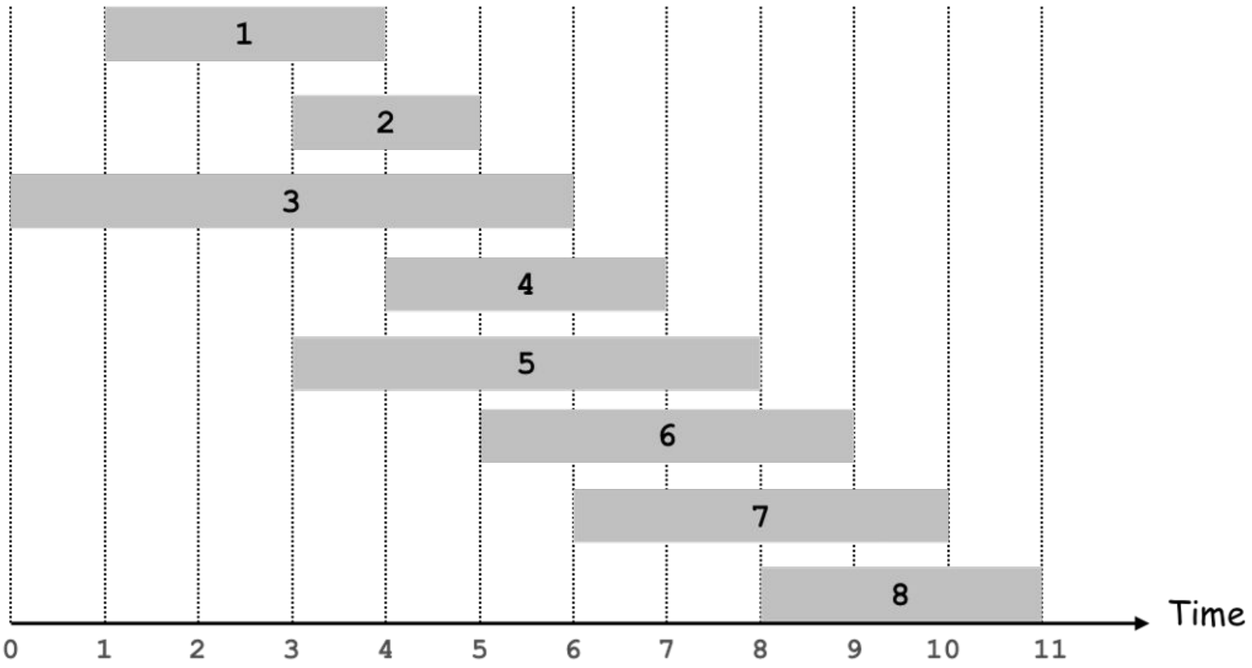
Weighted Interval Scheduling

- p(j) for an interval j
    - Returns the largest index i < j such that i and j are compatible
    - p(j) = 0 if no such interval exists




```
p(1)   = 0
p(2)?
       = 0
p(3)?
       = 1
p(4)?
       = 0
p(5)?
       = 3
p(6)?
       = 3
```

Weighted Interval Scheduling



p(1) = 0
p(2) = 0
p(3) = 0
p(4) = 1
p(5) = 0
p(6) = 2
p(7) = 3
p(8) = 5

Weighted Interval Scheduling

Let's consider an optimal solution `OPT` to any given set of input intervals

Property of `OPT`:

- Interval n (the last one), either belongs to `OPT` or it doesn't

If n ∈ `OPT`:

- no interval indexed between p(n) and n can belong to `OPT`
- `OPT` must include an optimal solution to {1, ..., p(n)}

If n ∉ `OPT`:

- `OPT` is simply equal to the optimal solution to the problem consisting of requests {1, ..., n-1}

Take It or Leave It

Let `OPT(i)` denote the max achievable value if we consider requests 1 through i

- `OPT(0) = 0`

Take it

- $\texttt{OPT(j) = } w_j \texttt{ + OPT(p(j))}$

Leave it

- `OPT(j) = OPT(j-1)`

DP Selection Principle

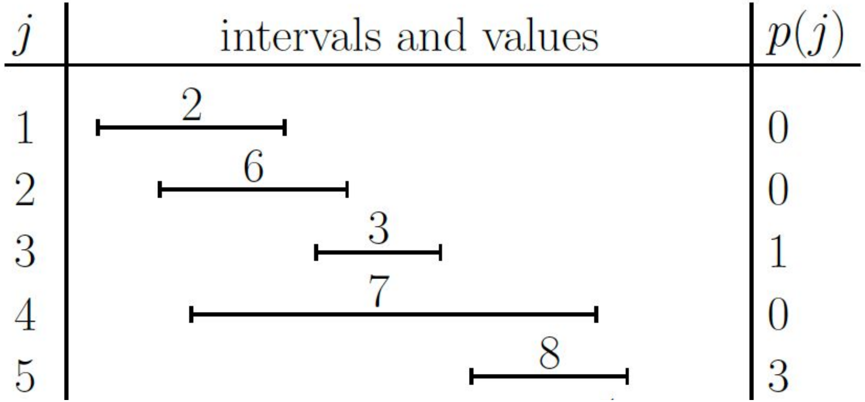
We either *take or leave* interval j

$$\mathtt{OPT(j)} = \max(\ w_j + \mathtt{OPT}(p(j)),\ \mathtt{OPT(j-1)}\ )$$

This can be written as a recursive function with base case:

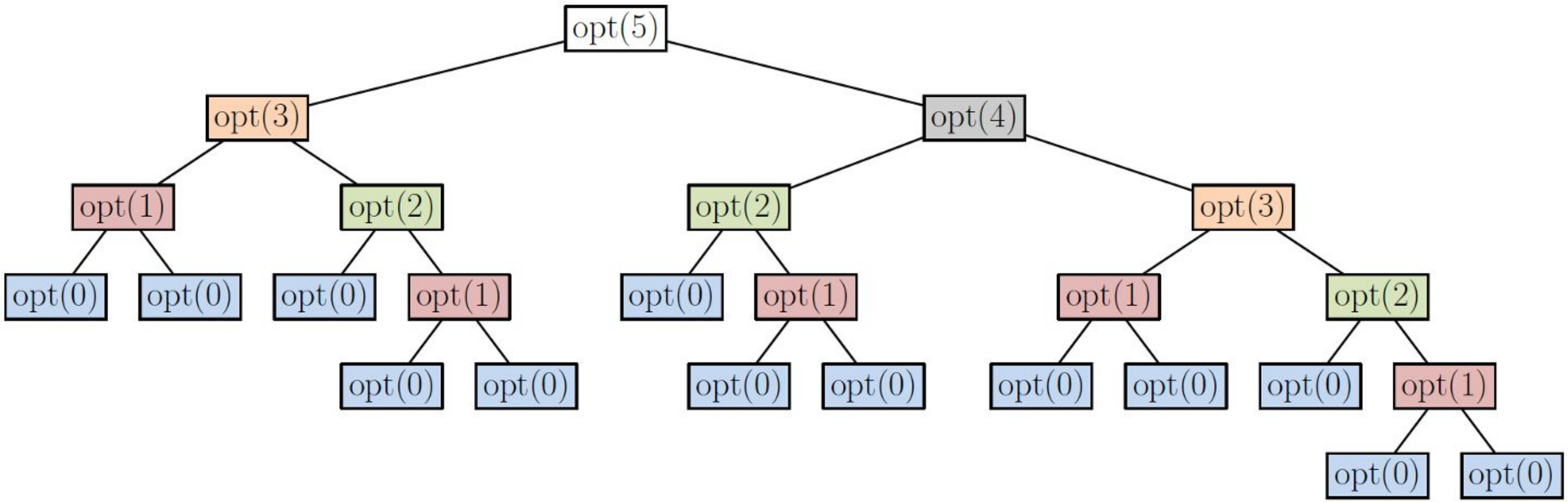
$$\mathtt{OPT(0)} = 0$$

DP Selection Principle




```
def OPT(j):
    if j==0: return 0
    return max( w_j + OPT(p(j)), OPT(j-1) )
```

Tree of Subproblems



Runtime of Recursive `OPT`

```
def OPT(j):
    if j==0: return 0
    return max( w_j + OPT(p(j)), OPT(j-1) )
```

How many calls do we make to OPT?

$T(j) = T(p(j)) + T(j-1) + O(1)$

$T(j) = T(j-1) + T(j-1) + O(1)$

$T(j) = 2*T(j-1)$

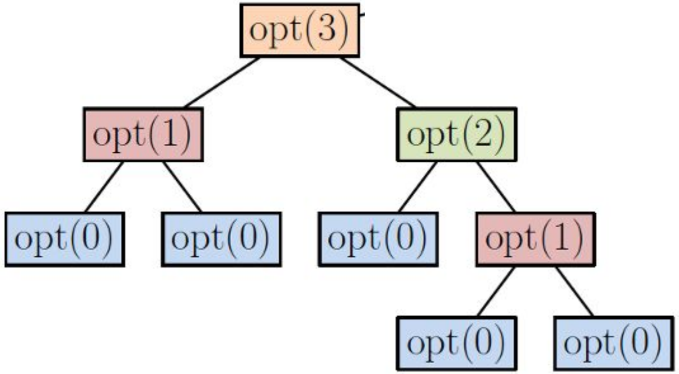
...

$O(2^n)$

Memoizing the Recursion

`OPT(n)` is really only solving n+1 subproblems, yet it makes exponentially many calls to `OPT`

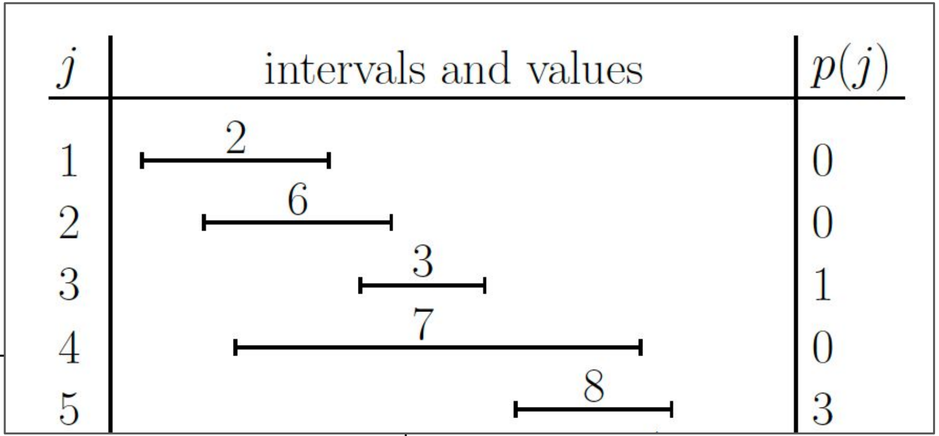
- redundant calls!



<u>Memoization</u>: store the results of each call in a global array

- Lookup rather than recomputing

Memoized OPT



Initialize global array M of size n to -1s

```
def M-OPT(j):
    if j==0: return 0
    else if M[j] != -1: return M[j]
    else:
        M[j] = max( w_j + M-OPT(p(j)), M-OPT(j-1) )
        return M[j]
```

Runtime Analysis

- memoization decreases the runtime, but by how much?

```
def M-OPT(j):
    if j==0: return 0
    else if M[j] != -1: return M[j]
    else:
        M[j] = max( w_j + M-OPT(p(j)), M-OPT(j-1) )
        return M[j]
```

Proof of Correctness - By Induction

$$opt(j) = \begin{cases} 0 \\ \max(opt(j-1), w_j + opt(p(j))) \end{cases}$$

Base Case: j = 0, `OPT`(0) = 0

IH: `OPT`(j) is correct for 0 ≤ j < n

Proof:

Case 1 (j ∉ S*) : `OPT(j) = OPT(j-1)`

correct by IH

Case 2 (j ∈ S*): `OPT(j) =` w_j `+ OPT( p(j) )`

Since j is selected, we cannot select any interval that overlaps with j

By definition of p, all intervals compatible with j must have index 0 ≤ p(j)

The other intervals we select must form an optimal solution to {1,2,..p(j)}

`OPT( p(j) )` correct by IH

Iterative version?

```
compute-opt(){
  M[0] = 0
  for (j = 1 to n) {
    if (M[j-1] > w[j]+M[p[j]]) {
      M[j] = M[j-1];
    }
    else {
      M[j] = w[j]+M[p[j]];
    }
  }
}
```

Runtime Analysis

- When we express it iteratively,

  Runtime is easier to analyze

  O(n)

```
compute-opt(){
  M[0] = 0
  for (j = 1 to n) {
    if (M[j-1] > w[j]+M[p[j]]) {
      M[j] = M[j-1];
    }
    else {
      M[j] = w[j]+M[p[j]];
    }
  }
}
```

Computing a Solution

```
M[0] = 0
for (j = 1 to n) {
  if (M[j-1] > w[j]+M[p[j]]) {
    M[j] = M[j-1]; pred[j] = j-1;
  }
  else {
    M[j] = w[j]+M[p[j]]; pred[j] = p[j];
  }
}
```

`pred`

0	0	0	2	0	3

DP Design

Break down the problem by **expressing the optimal solution in terms of optimal solutions to sub-parts**

- Either take or leave interval n

To do this, determine the recursive formulation

- $\max(w_i + OPT(p(i)), OPT(i-1))$

Summary

- HW5 due Monday
- Quiz on divide and conquer Nov 6
- Project Checkpoint 2 due tomorrow
- Lab 5 due tomorrow