Dynamic Programming

- Computing Fibonacci Numbers
- Weighted Interval Scheduling

Algorithmic Paradigms

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming History

Bellman. [1950s] Pioneered the systematic study of dynamic programming.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

> "it's impossible to use dynamic in a pejorative sense"
> "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ….

Some famous dynamic programming algorithms.

- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

Computing Fibonacci Number

Computing Fibonacci Numbers

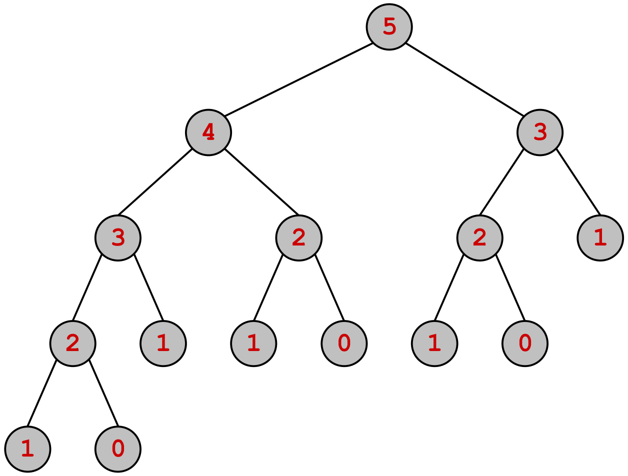
Fibonacci number: $F_n = F_{n-1}+F_{n-2}$, $F_0 = 0$ and $F_1 = 1$
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

Problem: Compute F_n for given n.

Example: if $n=8$, then $F_8=21$

Recursive Algorithm

```
Fib(n)
{
if n ≤ 1 then
   return 1
else
  return Fib(n-1)+Fib(n-2)
}
```



Running time: $T(n) = T(n-1) + T(n-2) \ \rightarrow \ T(n) = O\left((\frac{1+\sqrt{5}}{2})^n \right)$

Recursive Algorithm: Memoization

```
for j = 0 to n do
    F[j] = empty
F[0] =  0
F[1] = 1

Fib(n) {
if n ≤ 1 then
   return 1
else
  if F[n] = empty then
    F[n] = Fib(n-1)+Fib(n-2)
return F[n]
}
```

Running time: O(n)

- for each empty entry of F, we have two recursive calls and for each non-empty entry there is not any recursive call.
- At the beginning, we have n-1 empty entries and each time one is filled.

Non-recursive Algorithm (bottom-up iteration)

```
Fib(n)
{
F[0] = 0
F[1] = 1
for i = 2 to n  do
   F[i] = F[i-1]+F[i-2]
return F[n]
}
```

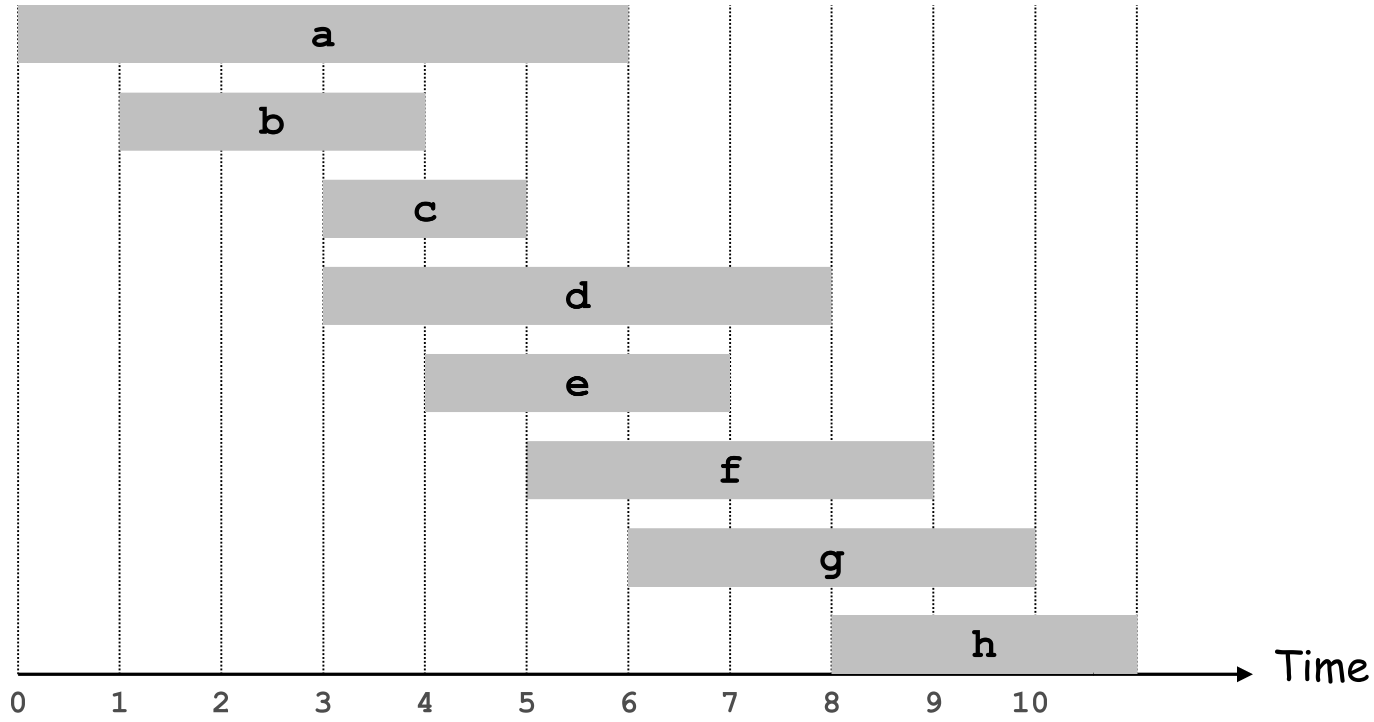
Running time: $O(n)$

Weighted Interval Scheduling

Weighted Interval Scheduling

Weighted interval scheduling problem.

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

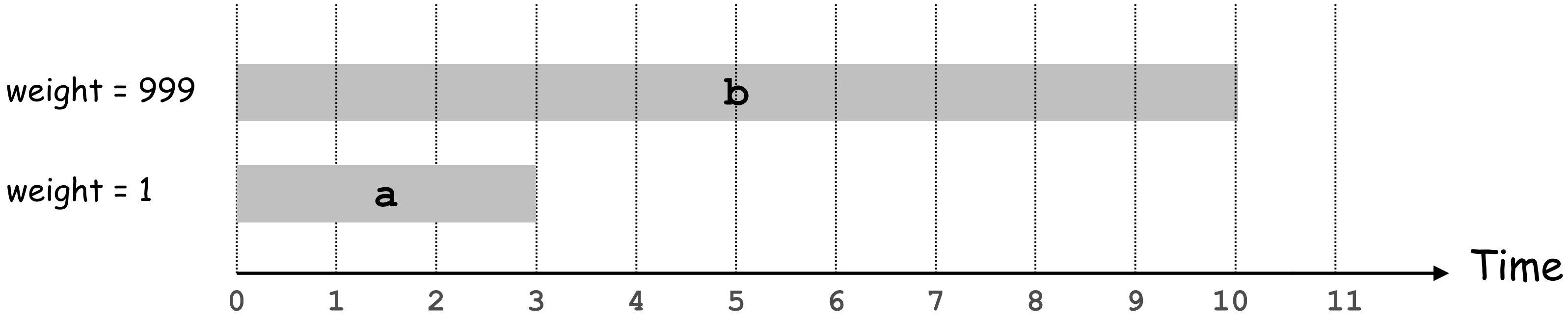


Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.

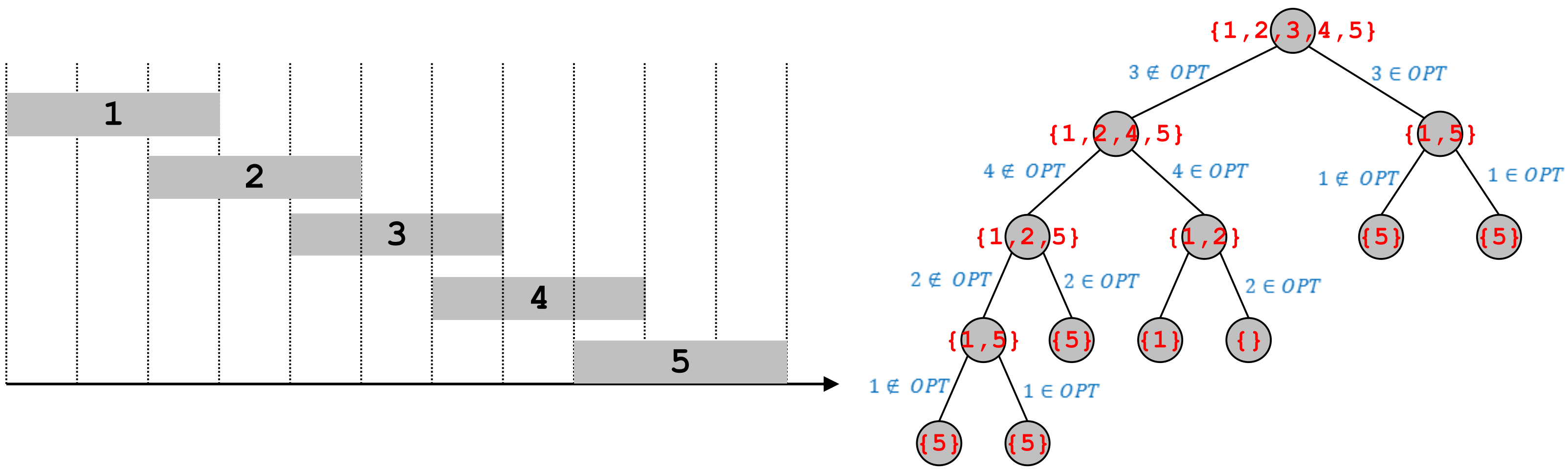
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



Weighted Interval Scheduling: Brute Force

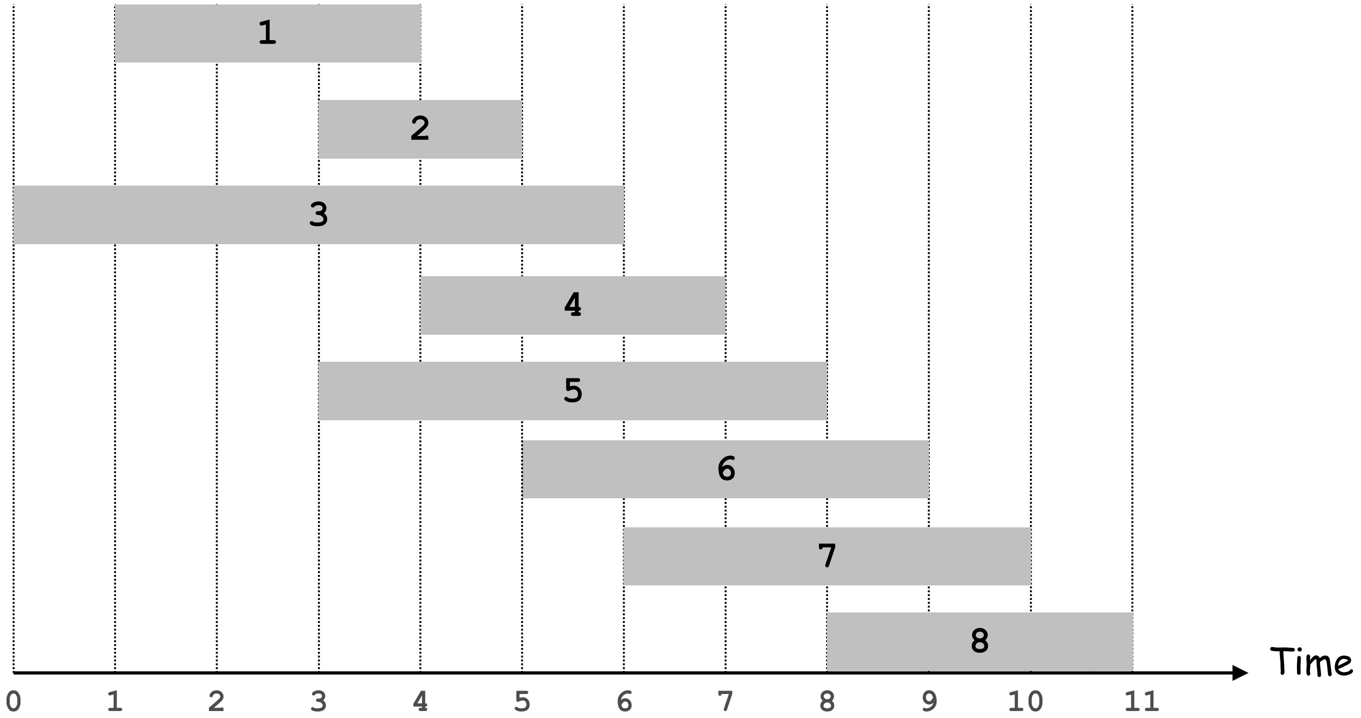
Observation. Recursive algorithm fails spectacularly because of too many sub-problems $\Rightarrow$ exponential algorithms.



- Sub-problems {1,5} and {1,2} are different although they have the same size
- The order in which we check jobs whether they are in OPT is in our hand. So select an order which produces fewer sub-problems.

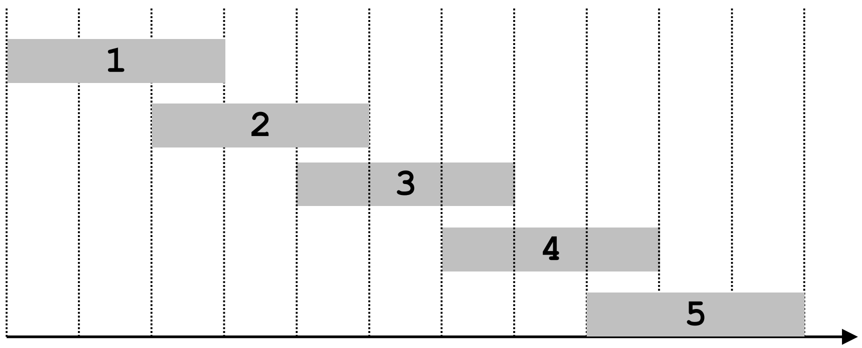
Weighted Interval Scheduling

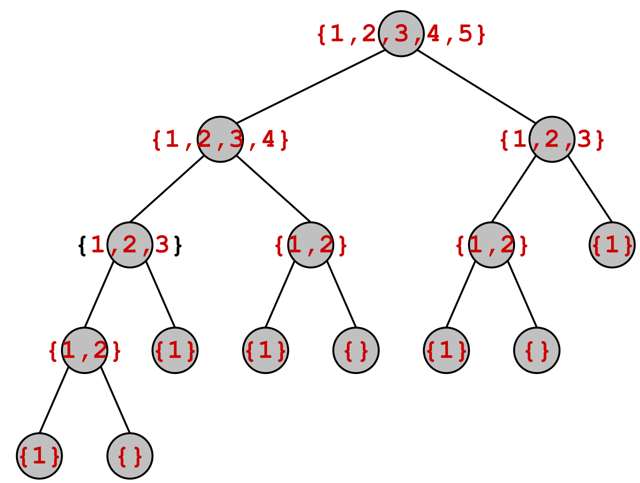
Observation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.



Weighted Interval Scheduling: Brute Force

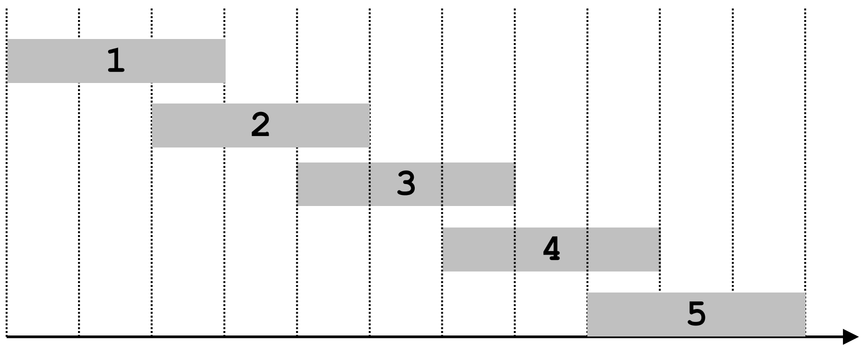
Observation. Sub-problems are of form {1,2, …,i}. We have n such sub-problems. Let's show {1,2, …, i} by i

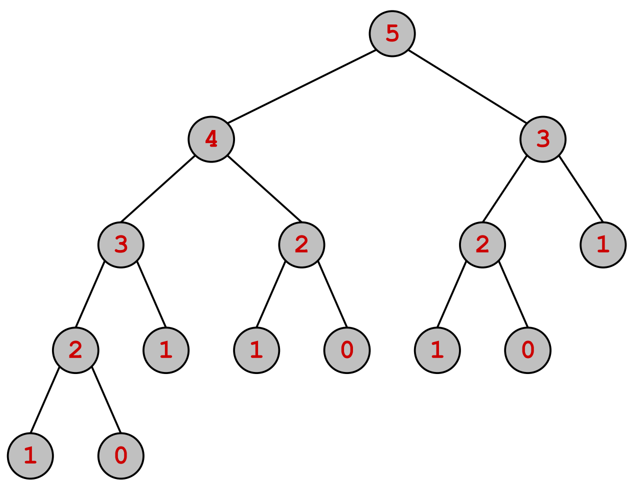




Weighted Interval Scheduling: Brute Force

Observation. Sub-problems are of form {1,2, …,i}. We have n such sub-problems. Let's show {1,2, …, i} by i

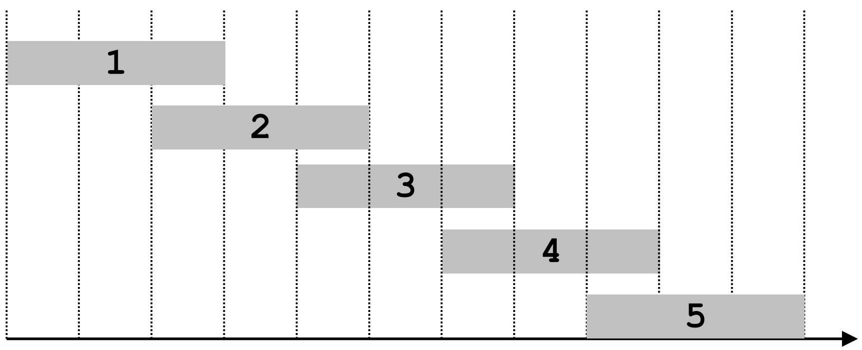


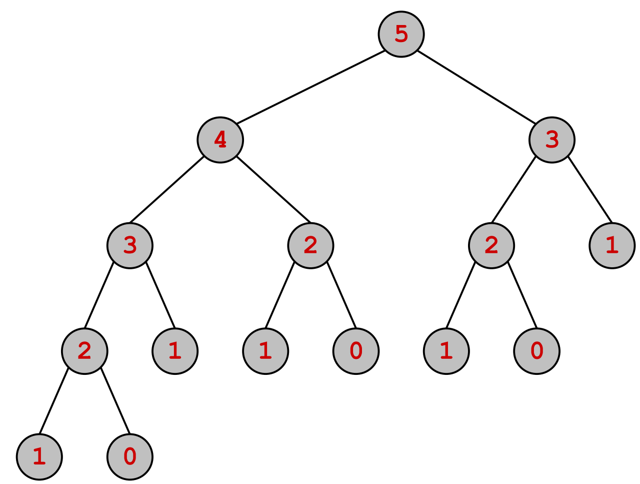


Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm still fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.

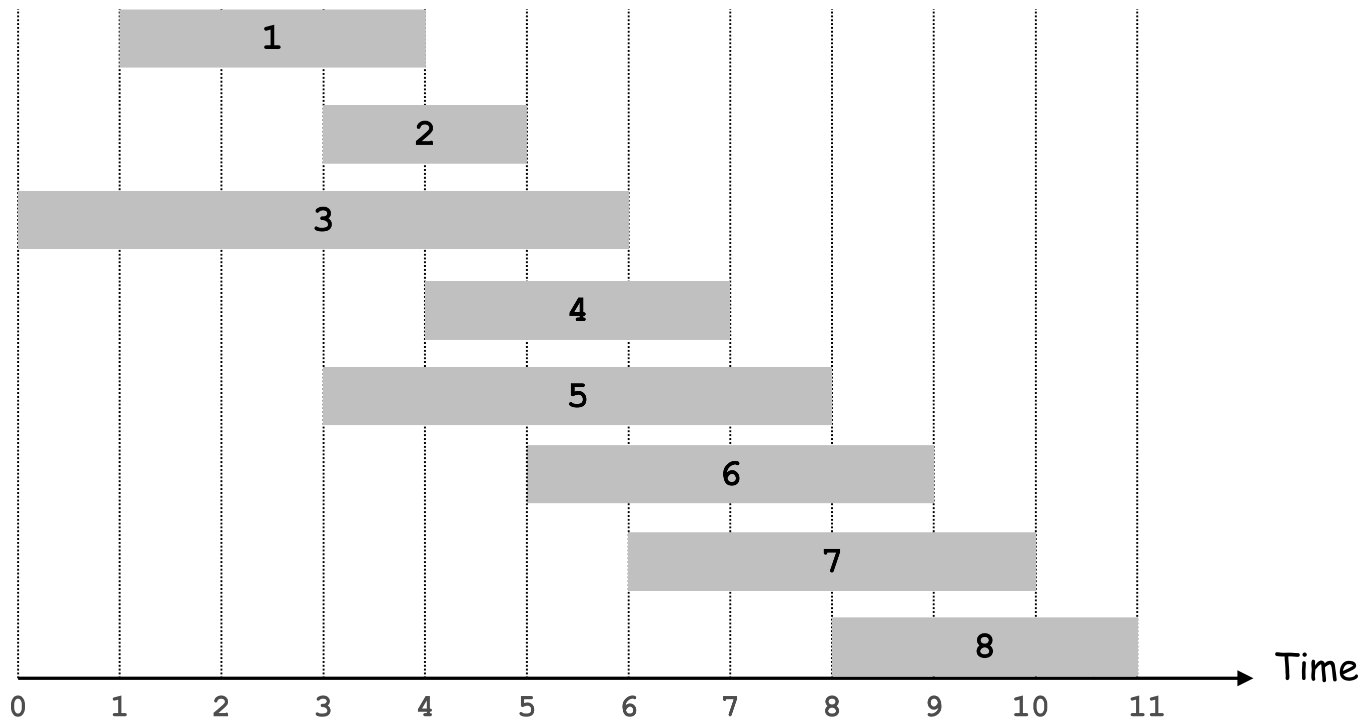




Weighted Interval Scheduling

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - collect profit v_j
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\ \{\ v_j + OPT(p(j)),\ OPT(j-1)\ \} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Brute Force

Brute force algorithm.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty          global array
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.

- `M-Compute-Opt(j)`: each invocation takes $O(1)$ time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

- Progress measure Φ = # nonempty entries of `M[]`.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases Φ by 1 $\Rightarrow$ at most $2n$ recursive calls.

- Overall running time of `M-Compute-Opt(n)` is $O(n)$. ▪

Remark. $O(n)$ if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (vj + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

- # of recursive calls $\leq n \Rightarrow O(n)$.

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(v_j + M[p(j)], M[j-1])
}
```

Dynamic Programming Algorithm: Summary

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution in a bottom-up fashion
- Construct an optimal solution from the computed information (not always necessary)

References

References

- Sections 6.1-2 of the text book "algorithm design" by Jon Kleinberg and Eva Tardos
- The original slides were prepared by Kevin Wayne. The slides are distributed by Pearson Addison-Wesley.