Lecture 12: Dynamic Programming I

Michael Dinitz

October 2, 2025

601.433/633 Introduction to Algorithms

Introduction

Dynamic Programming: divide and conquer++

Classical divide and conquer (quicksort, mergesort, . . .)

- Divide problem into subproblems
- Solve each subproblem
- Combine solutions from subproblems into solution for problem
- Usually implemented with recursion

Issues that dynamic programming can help with:

- What if subproblems *overlap*?
- What if recursion too slow?

Today: motivate dynamic programming through simple example
~~Thursday~~: more complicated examples
Tuesday

Notes

Dynamic programming used all over the place

- Originally in control theory
- Then many uses in graph algorithms, combinatorial optimization
- Currently: many uses in strings

At JHU:

- String algorithms: NLP!
  - Jason Eisner: new programming language *Dyna* to *automatically* do dynamic programming
- String algorithms: computational biology!

Why "Dynamic Programming": Richard Bellman

An interesting question is, Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

Example: Weighted Interval Scheduling

Weighted Interval Scheduling: Definition

Input:

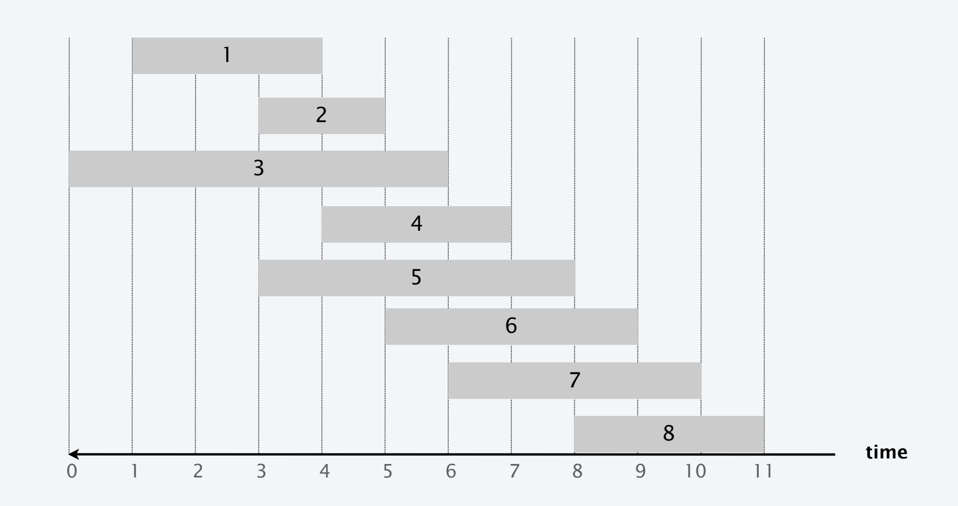
- n requests (intervals) $\{1, 2, \ldots, n\}$
- For each request i:
  - Start time s_i
  - Finish time f_i
  - Value v_i
- Assume sorted by finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$

Feasible:

- $S \subseteq [n]$ feasible if no two intervals of S overlap
  - $(s_i, f_i) \cap (s_j, f_j) = \emptyset$ for all $i, j \in S$ with $i \neq j$

Goal:

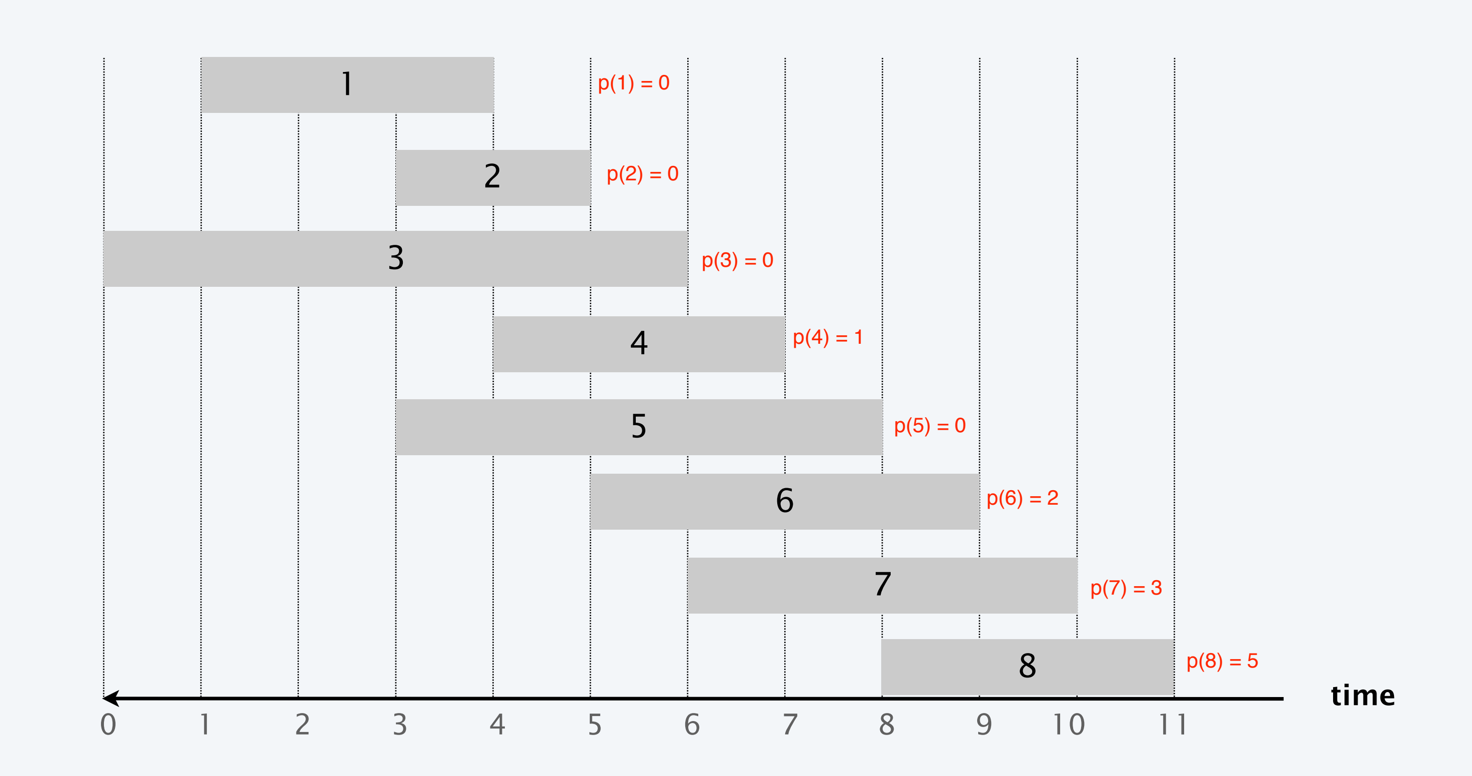
- Find feasible S maximizing $v(S) = \sum_{i \in S} v_i$



Definition II

Definition

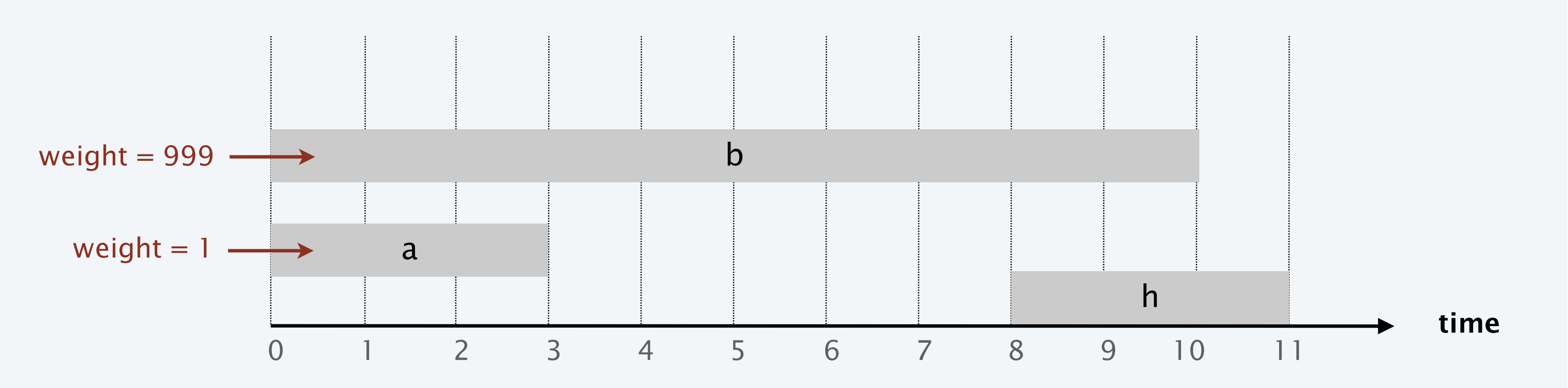
Let $p(i)$ largest $j < i$ such that $f_j \leq s_i$. If no such j exists, $p(i) = 0$.



Obvious Approach

Obvious Approach

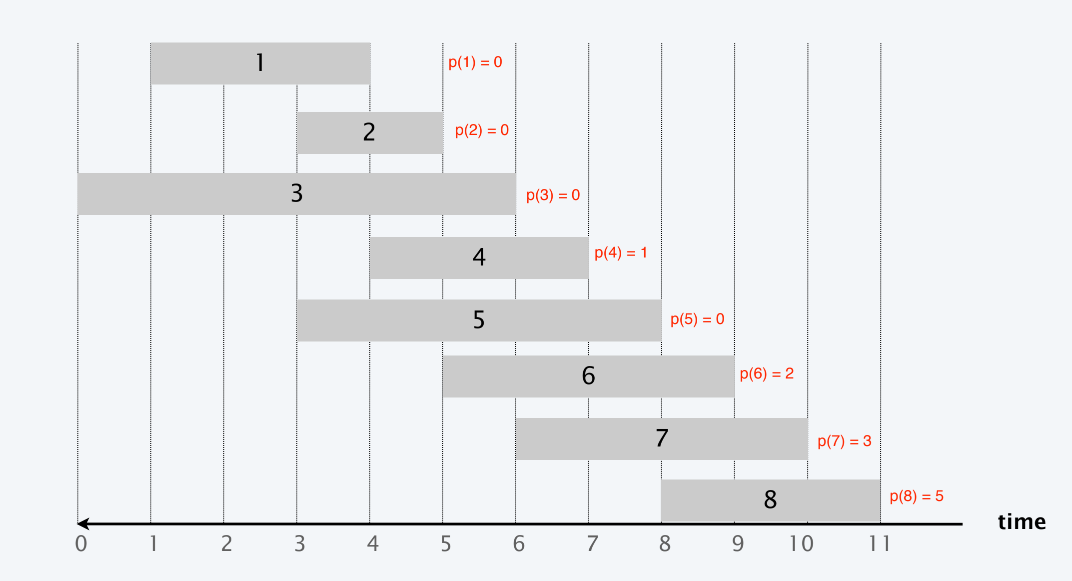
No variation of greedy works.
Example: greedy by earliest finishing times



Need fundamentally different approach

Simple Observation

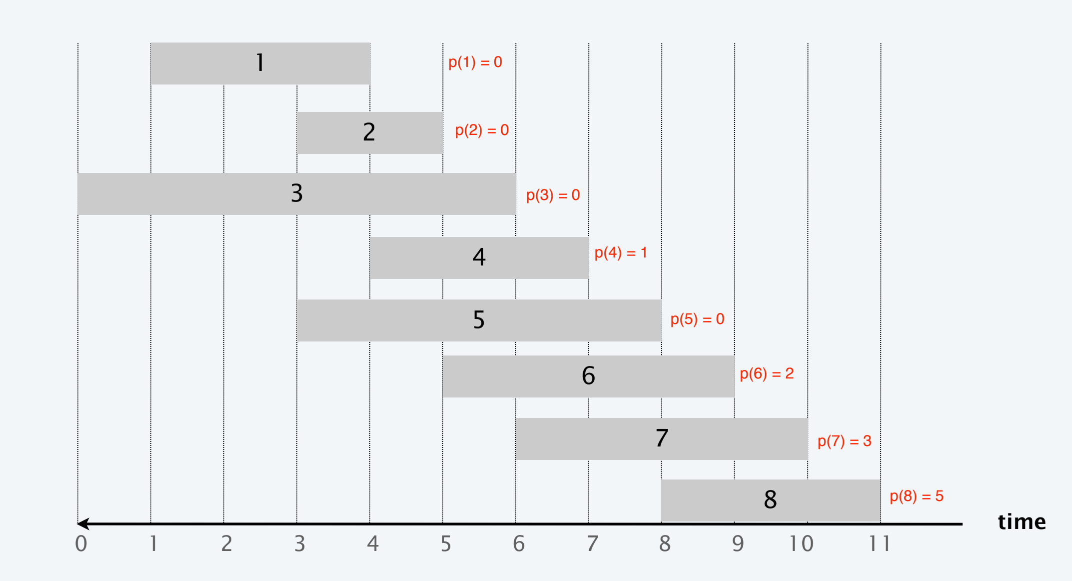
Let $S^* \subseteq [n]$ be optimal solution (unknown).
What simple observation can we make about S^*?



Simple Observation

Let $S^* \subseteq [n]$ be optimal solution (unknown).
What simple observation can we make about S^*?

Fact: Either $n \in S^*$ or $n \notin S^*$

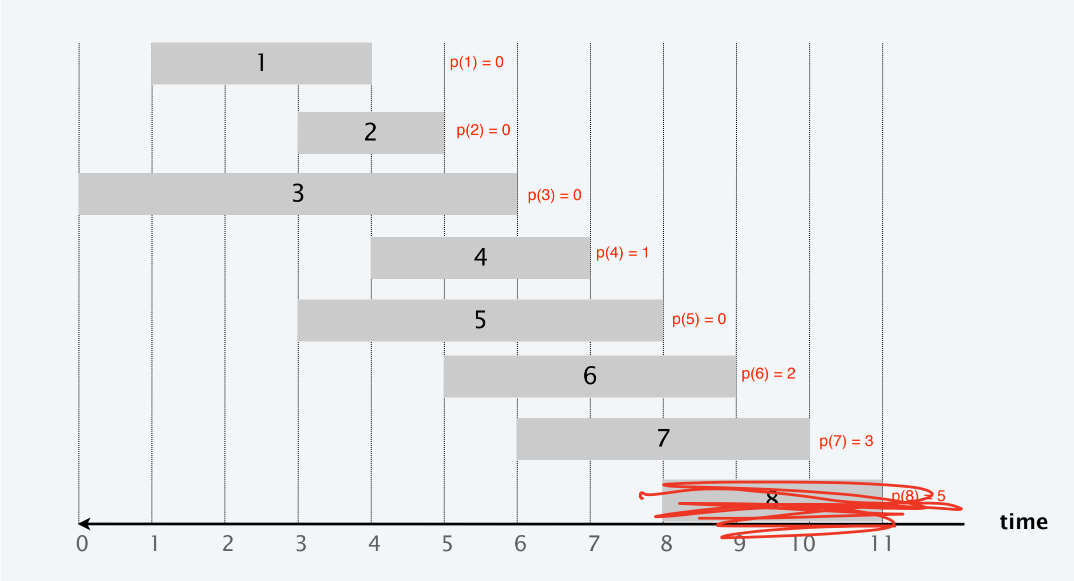


Simple Observation

Let $S^* \subseteq [n]$ be optimal solution (unknown).
What simple observation can we make about S^*?

Fact: Either $n \in S^*$ or $n \notin S^*$

If $n \notin S^*$:

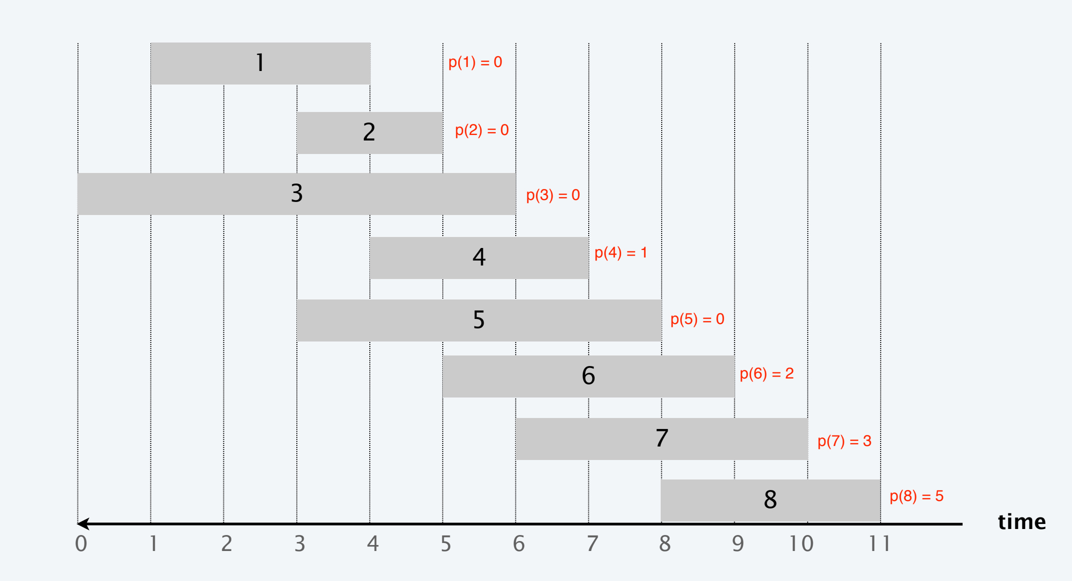


Simple Observation

Let $S^* \subseteq [n]$ be optimal solution (unknown).
What simple observation can we make about S^*?

Fact: Either $n \in S^*$ or $n \notin S^*$

If $n \notin S^*$: S^* optimal solution for $\{1, 2, \ldots, n-1\}$



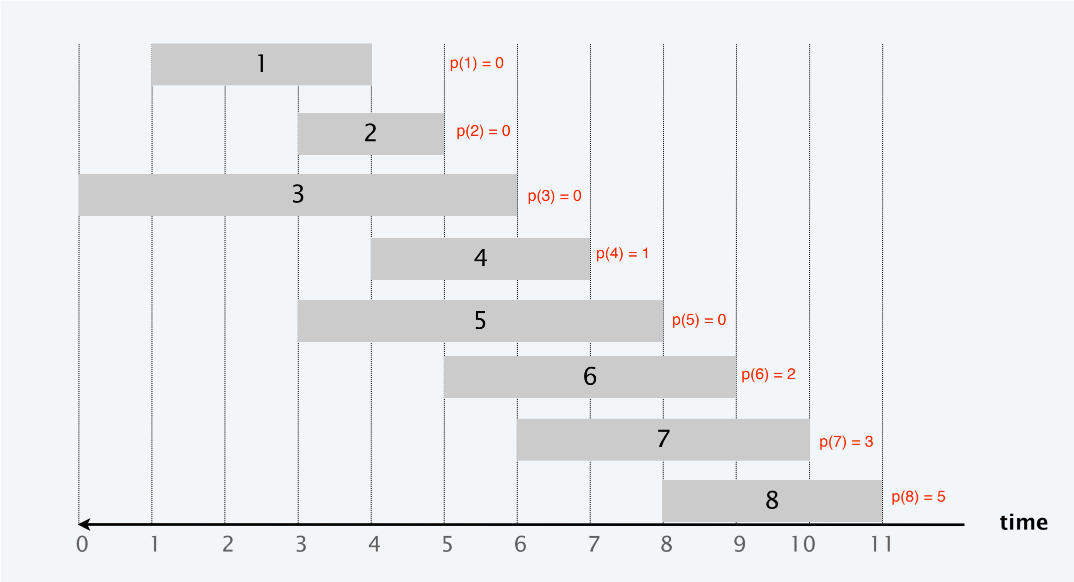
Simple Observation

Let $S^* \subseteq [n]$ be optimal solution (unknown).
What simple observation can we make about S^*?

Fact: Either $n \in S^*$ or $n \notin S^*$

If $n \notin S^*$: S^* optimal solution for $\{1, 2, \ldots, n-1\}$

If $n \in S^*$:



Simple Observation

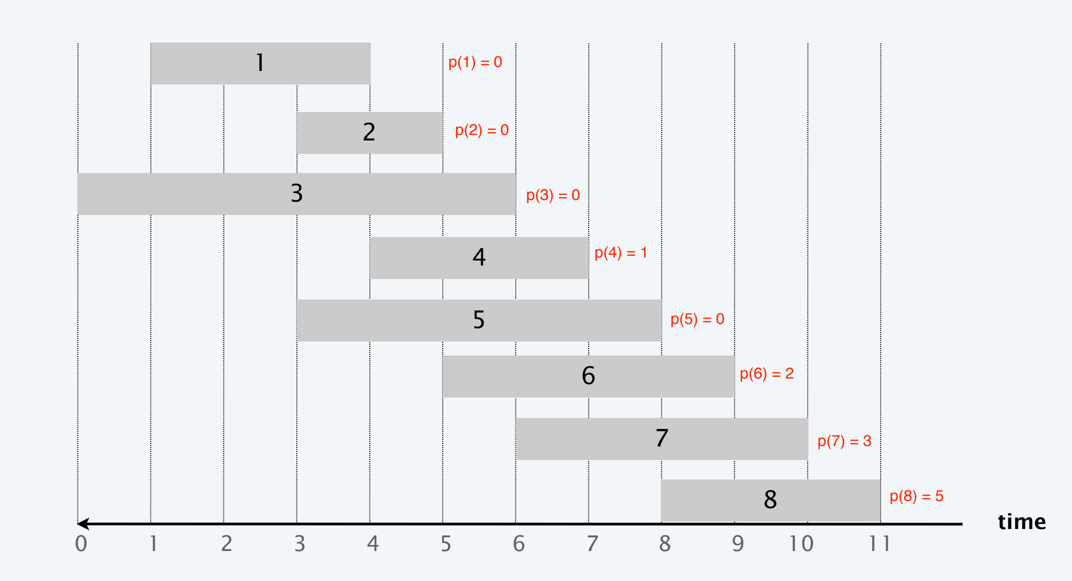
Let $S^* \subseteq [n]$ be optimal solution (unknown).
What simple observation can we make about S^*?

Fact: Either $n \in S^*$ or $n \notin S^*$

If $n \notin S^*$: S^* optimal solution for $\{1, 2, \ldots, n-1\}$

If $n \in S^*$:

- Nothing in $(p(n), n-1]$ in S^*: overlap with n



Simple Observation

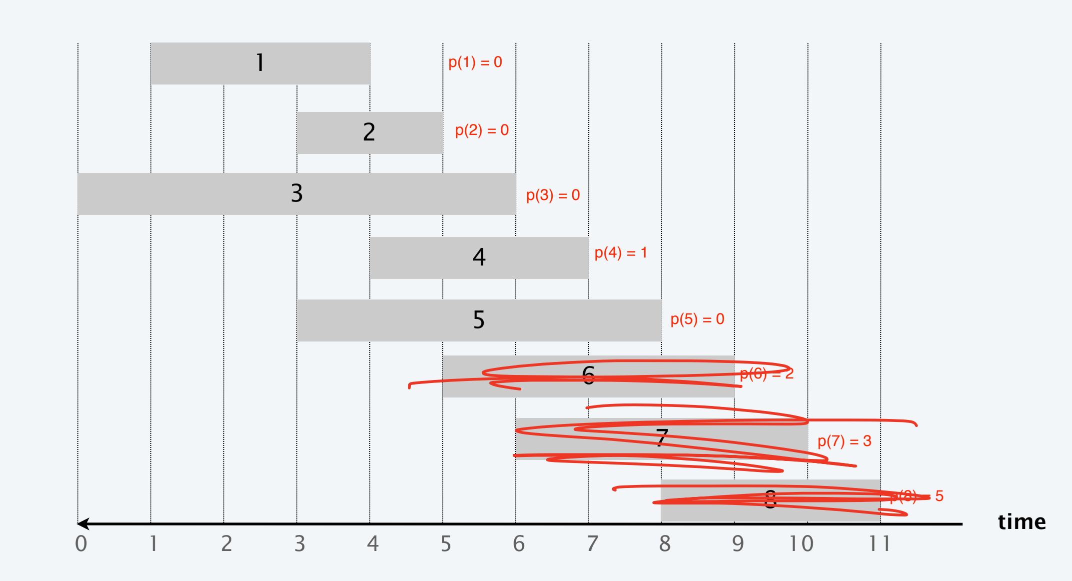
Let $S^* \subseteq [n]$ be optimal solution (unknown).
What simple observation can we make about S^*?

Fact: Either $n \in S^*$ or $n \notin S^*$

If $n \notin S^*$: S^* optimal solution for $\{1, 2, \ldots, n-1\}$

If $n \in S^*$:

- Nothing in $(p(n), n-1]$ in S^*: overlap with n
- $S^* = \{n\} \cup$ opt solution for $\{1, 2, \ldots, p(n)\}$



Formalize

Definition

Let $OPT(i)$ denote *value* of optimal solution S_i^* for $\{1, 2, \ldots, i\}$

Note:

- S_i^* not necessarily equal to $S^* \cap \{1, 2, \ldots, i\}$ (but $S_n^* = S^*$)
- $OPT(0) = 0$ by convention

Formalize

Definition

Let $OPT(i)$ denote *value* of optimal solution S_i^* for $\{1, 2, \ldots, i\}$

Note:

- S_i^* not necessarily equal to $S^* \cap \{1, 2, \ldots, i\}$ (but $S_n^* = S^*$)
- $OPT(0) = 0$ by convention

If $n \notin S^*$: $OPT(n) =$

Formalize

Definition

Let $OPT(i)$ denote *value* of optimal solution S_i^* for $\{1, 2, \ldots, i\}$

Note:

- S_i^* not necessarily equal to $S^* \cap \{1, 2, \ldots, i\}$ (but $S_n^* = S^*$)
- $OPT(0) = 0$ by convention

If $n \notin S^*$: $OPT(n) = OPT(n-1)$

Formalize

Definition

Let $OPT(i)$ denote *value* of optimal solution S_i^* for $\{1, 2, \ldots, i\}$

Note:

- S_i^* not necessarily equal to $S^* \cap \{1, 2, \ldots, i\}$ (but $S_n^* = S^*$)
- $OPT(0) = 0$ by convention

If $n \notin S^*$: $OPT(n) = OPT(n-1)$
If $n \in S^*$: $OPT(n) =$

Formalize

Definition

Let $OPT(i)$ denote *value* of optimal solution S_i^* for $\{1, 2, \ldots, i\}$

Note:

- S_i^* not necessarily equal to $S^* \cap \{1, 2, \ldots, i\}$ (but $S_n^* = S^*$)
- $OPT(0) = 0$ by convention

If $n \notin S^*$: $OPT(n) = OPT(n-1)$
If $n \in S^*$: $OPT(n) = v_n + OPT(p(n))$

Formalize

Definition

Let $OPT(i)$ denote *value* of optimal solution S_i^* for $\{1, 2, \ldots, i\}$

Note:

- S_i^* not necessarily equal to $S^* \cap \{1, 2, \ldots, i\}$ (but $S_n^* = S^*$)
- $OPT(0) = 0$ by convention

If $n \notin S^*$: $OPT(n) = OPT(n-1)$
If $n \in S^*$: $OPT(n) = v_n + OPT(p(n))$

Don't know if $n \in S^*$, but can still say:

$$OPT(n) =$$

Formalize

Definition

Let $OPT(i)$ denote *value* of optimal solution S_i^* for $\{1, 2, \ldots, i\}$

Note:

- S_i^* not necessarily equal to $S^* \cap \{1, 2, \ldots, i\}$ (but $S_n^* = S^*$)
- $OPT(0) = 0$ by convention

If $n \notin S^*$: $OPT(n) = OPT(n-1)$
If $n \in S^*$: $OPT(n) = v_n + OPT(p(n))$

Don't know if $n \in S^*$, but can still say:

$$OPT(n) = \max(OPT(n-1), v_n + OPT(p(n)))$$

Formalize

Definition

Let $OPT(i)$ denote *value* of optimal solution S_i^* for $\{1, 2, \ldots, i\}$

Note:

- S_i^* not necessarily equal to $S^* \cap \{1, 2, \ldots, i\}$ (but $S_n^* = S^*$)
- $OPT(0) = 0$ by convention

If $n \notin S^*$: $OPT(n) = OPT(n-1)$
If $n \in S^*$: $OPT(n) = v_n + OPT(p(n))$

Don't know if $n \in S^*$, but can still say:

$$OPT(n) = \max(OPT(n-1), v_n + OPT(p(n)))$$

Now need to prove this more formally...

Structure Theorem

Theorem

$\boldsymbol{OPT(j) = \max(OPT(j-1), v_j + OPT(p(j)))}$ *for all* $\boldsymbol{1 \leq j \leq n}$

Structure Theorem

Theorem

$OPT(j) = \max(OPT(j-1), v_j + OPT(p(j)))$ *for all* $1 \le j \le n$

$\geq$: Know there are feasible solutions to $\{1, 2, \ldots, j\}$ of value:

- $OPT(j-1)$ (S^*_{j-1} feasible for $\{1, 2, \ldots, j\}$)
- $v_j + OPT(p(j))$ (add j to $S^*_{p(j)}$)

$\implies OPT(j) \geq \max(OPT(j-1), v_j + OPT(p(j)))$

Structure Theorem

Theorem

$OPT(j) = \max(OPT(j-1), v_j + OPT(p(j)))$ *for all* $1 \le j \le n$

$\ge$: Know there are feasible solutions to $\{1, 2, \ldots, j\}$ of value:

- $OPT(j-1)$ (S^*_{j-1} feasible for $\{1, 2, \ldots, j\}$)
- $v_j + OPT(p(j))$ (add j to $S^*_{p(j)}$)

$\implies OPT(j) \ge \max(OPT(j-1), v_j + OPT(p(j)))$

$\le$: Two cases

Structure Theorem

Theorem

$OPT(j) = \max(OPT(j-1), v_j + OPT(p(j)))$ *for all* $1 \le j \le n$

$\ge$: Know there are feasible solutions to $\{1, 2, \ldots, j\}$ of value:

- $OPT(j-1)$ (S^*_{j-1} feasible for $\{1, 2, \ldots, j\}$)
- $v_j + OPT(p(j))$ (add j to $S^*_{p(j)}$)

$\implies OPT(j) \ge \max(OPT(j-1), v_j + OPT(p(j)))$

$\le$: Two cases

- If $j \notin S^*_j$, then $S^*_j \subseteq \{1, 2, \ldots, j-1\}$

$\implies S^*_j$ feasible for $[j-1] \implies OPT(j) \le OPT(j-1)$ (definition of $OPT(j-1)$)

Structure Theorem

Theorem

$OPT(j) = \max(OPT(j-1), v_j + OPT(p(j)))$ *for all* $1 \le j \le n$

$\geq$: Know there are feasible solutions to $\{1, 2, \ldots, j\}$ of value:

- $OPT(j-1)$ (S^*_{j-1} feasible for $\{1, 2, \ldots, j\}$)
- $v_j + OPT(p(j))$ (add j to $S^*_{p(j)}$)

$\implies OPT(j) \geq \max(OPT(j-1), v_j + OPT(p(j)))$

$\leq$: Two cases

- If $j \notin S^*_j$, then $S^*_j \subseteq \{1, 2, \ldots, j-1\}$

  $\implies S^*_j$ feasible for $[j-1] \implies OPT(j) \leq OPT(j-1)$ (definition of $OPT(j-1)$)
- If $j \in S^*_j$, then by definition $S^*_j \setminus \{j\}$ feasible for $\{1, 2, \ldots, p(j)\}$

  $\implies OPT(j) - v_j = v(S^*_j \setminus \{j\}) \leq OPT(p(j))$ (def of $OPT(p(j))$)

  $\implies OPT(j) \leq OPT(p(j)) + v_j$.

Obvious Algorithm

Previous theorem a recurrence relation!

- Suggests obvious recursive algorithm for computing $\boldsymbol{OPT(j)}$

Obvious Algorithm

Previous theorem a recurrence relation!

- Suggests obvious recursive algorithm for computing $\boldsymbol{OPT(j)}$

Schedule(j) {
  If $\boldsymbol{j = 0}$ return $\mathbf{0}$;
  else return max(Schedule($\boldsymbol{j-1}$), $\boldsymbol{v_j}$ + Schedule($\boldsymbol{p(j)}$));
}

Correctness

Theorem

Schedule($\boldsymbol{j}$) returns $\boldsymbol{OPT(j)}$.

Correctness

Theorem

Schedule($\boldsymbol{j}$) *returns* $\boldsymbol{OPT(j)}$.

Proof.

Induction on $\boldsymbol{j}$

Correctness

Theorem

Schedule(j) returns $OPT(j)$.

Proof.

Induction on j

- Base case: $j = 0$. Then Schedule(j) returns $0 = OPT(j)$

Correctness

Theorem

Schedule(j) returns $OPT(j)$.

Proof.

Induction on j

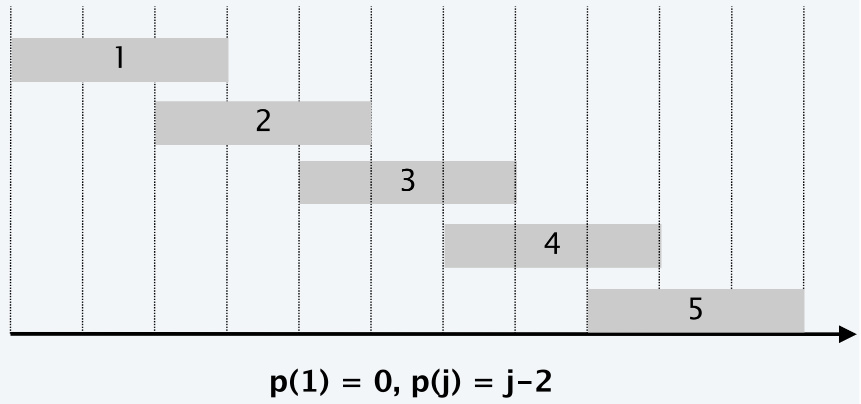
- Base case: $j = 0$. Then Schedule(j) returns $0 = OPT(j)$
- Inductive step: Schedule(j) returns

$$\begin{aligned} &\max(\text{Schedule}(j-1),\ v_j + \text{Schedule}(p(j))) && \text{(def of algorithm)} \\ &= \max(OPT(j-1), v_j + OPT(p(j))) && \text{(induction)} \\ &= OPT(j) && \text{(structure theorem)} \end{aligned}$$

□

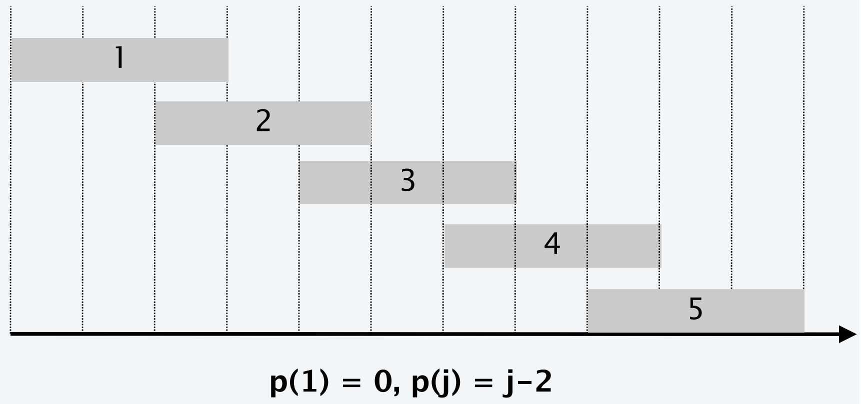
Running Time

Suppose $\boldsymbol{p(j) = j - 2}$ for all $\boldsymbol{j}$:



Running Time

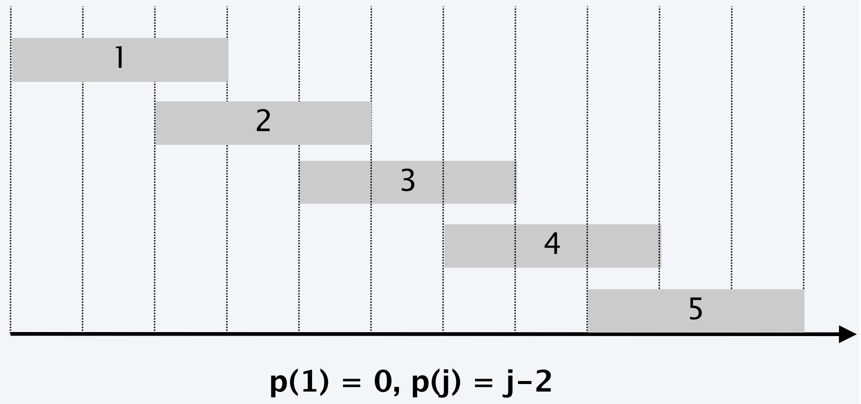
Suppose $\boldsymbol{p(j) = j - 2}$ for all $\boldsymbol{j}$:

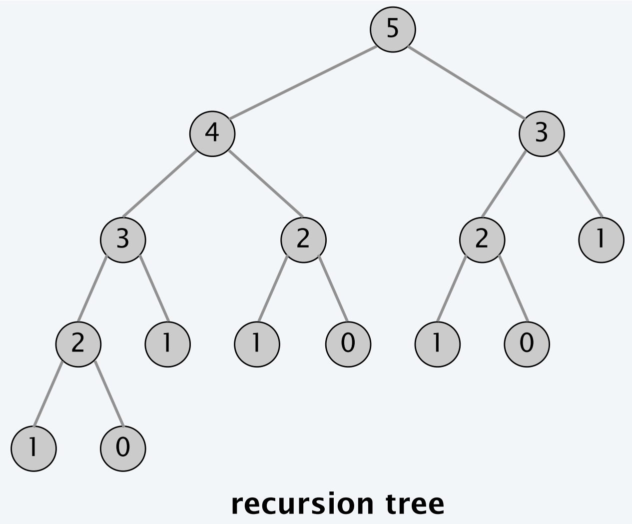


Schedule($\boldsymbol{j}$) calls Schedule($\boldsymbol{j - 1}$) and Schedule($\boldsymbol{j - 2}$)

Running Time

Suppose $\boldsymbol{p(j) = j - 2}$ for all $\boldsymbol{j}$:

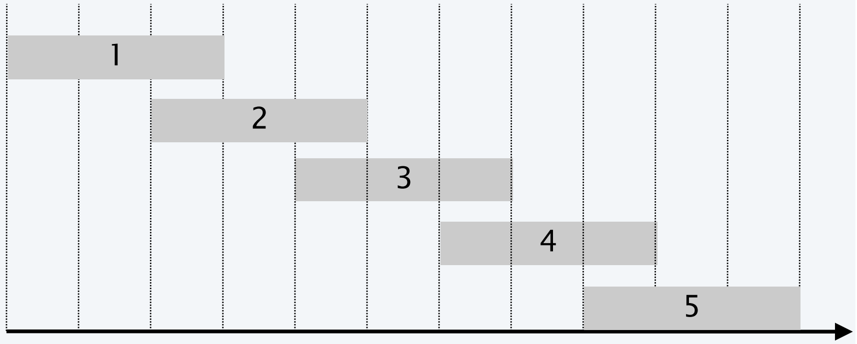




Schedule($\boldsymbol{j}$) calls Schedule($\boldsymbol{j-1}$) and Schedule($\boldsymbol{j-2}$)

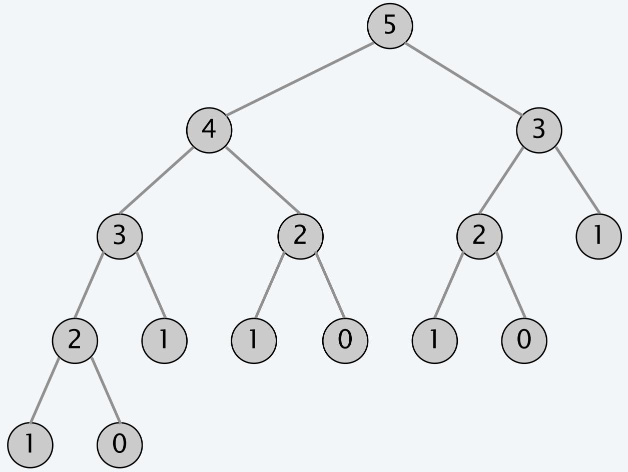
Running Time

Suppose $p(j) = j - 2$ for all j:



p(1) = 0, p(j) = j–2

Schedule(j) calls Schedule($j - 1$) and Schedule($j - 2$)



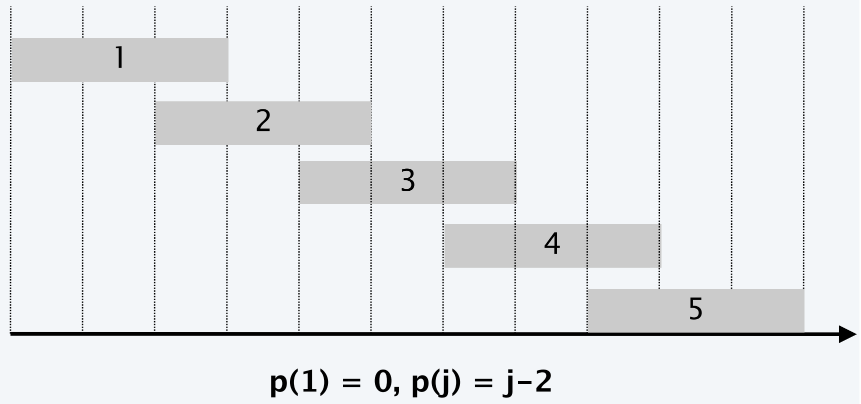
recursion tree

Let $T(n)$ be running time of Schedule(n) on this instance

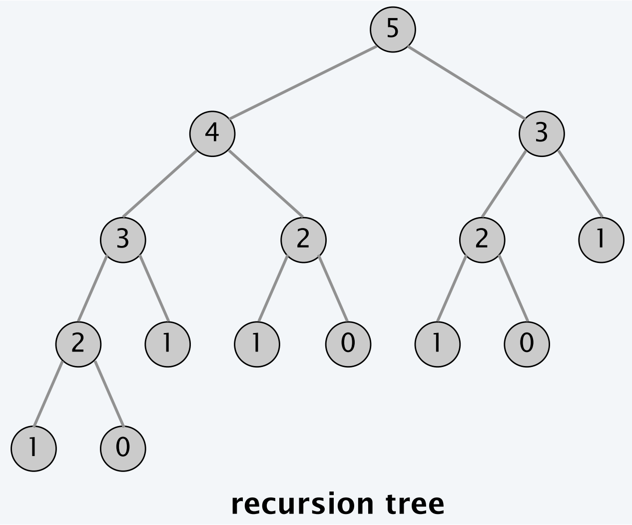
$$T(n) = T(n-1) + T(n-2) + c$$

Running Time

Suppose $p(j) = j - 2$ for all j:



Schedule(j) calls Schedule($j - 1$) and Schedule($j - 2$)



Let $T(n)$ be running time of Schedule(n) on this instance

$$T(n) = T(n-1) + T(n-2) + c$$

Fibonacci numbers: exponential in n

Fix: Memoization

Idea: avoid recomputation!

Fix: Memoization

Idea: avoid recomputation!

Table $\boldsymbol{M}$ of size $\boldsymbol{n}$, initially all empty

Fix: Memoization

Idea: avoid recomputation!

Table $\boldsymbol{M}$ of size $\boldsymbol{n}$, initially all empty

Schedule(j) $\{$

If $\boldsymbol{j} = \mathbf{0}$ return $\mathbf{0}$;

else if $\boldsymbol{M}[\boldsymbol{j}]$ nonempty return $\boldsymbol{M}[\boldsymbol{j}]$;

else $\{$

$\boldsymbol{M}[\boldsymbol{j}] = \mathbf{max}(\text{Schedule}(\boldsymbol{j}-\mathbf{1}), \boldsymbol{v_j} + \text{Schedule}(\boldsymbol{p}(\boldsymbol{j})))$;

return $\boldsymbol{M}[\boldsymbol{j}]$;

$\}$

$\}$

Fix: Memoization

Idea: avoid recomputation!

Table M of size n, initially all empty

```
Schedule(j) {
    If j = 0 return 0;
    else if M[j] nonempty return M[j];
    else {
        M[j] = max(Schedule(j − 1), v_j + Schedule(p(j)));
        return M[j];
    }
}
```

Correctness: (basically) same as before.

- Change inductive hypothesis to: "Schedule(j) returns $OPT(j)$ and after it returns, $M[j] = OPT(j)$"

Running Time

Theorem

The worst-case running time of Schedule($\boldsymbol{n}$) is at most $\boldsymbol{O(n)}$.

Running Time

Theorem

The worst-case running time of Schedule($\boldsymbol{n}$) is at most $\boldsymbol{O(n)}$.

Proof.

On call to Schedule($\boldsymbol{j}$):

- Either return entry from table ($\boldsymbol{O(1)}$ time), or
- Two recursive calls, then fill in table entry that was empty

Running Time

Theorem

The worst-case running time of Schedule(n) is at most $O(n)$.

Proof.

On call to Schedule(j):

- Either return entry from table ($O(1)$ time), or
- Two recursive calls, then fill in table entry that was empty

$\Longrightarrow$ running time = $O(1) \times \#$ recursive calls

Running Time

Theorem

The worst-case running time of Schedule(n) is at most $O(n)$.

Proof.

On call to Schedule(j):

- Either return entry from table ($O(1)$ time), or
- Two recursive calls, then fill in table entry that was empty

$\implies$ running time = $O(1) \times \#$ recursive calls

Fill in one (previously empty) table entry after 2 recursive calls
$\implies$ At most $2n$ recursive calls

Running Time

Theorem

The worst-case running time of Schedule(n) is at most $O(n)$.

Proof.

On call to Schedule(j):

- Either return entry from table ($O(1)$ time), or
- Two recursive calls, then fill in table entry that was empty

$\implies$ running time = $O(1) \times \#$ recursive calls

Fill in one (previously empty) table entry after 2 recursive calls
$\implies$ At most $2n$ recursive calls

So running time at most $O(n)$ □

Running Time

Theorem

The worst-case running time of Schedule(n) is at most $O(n)$.

Proof.

On call to Schedule(j):

- Either return entry from table ($O(1)$ time), or
- Two recursive calls, then fill in table entry that was empty

$\implies$ running time = $O(1) \times \#$ recursive calls

Fill in one (previously empty) table entry after 2 recursive calls
$\implies$ At most $2n$ recursive calls

So running time at most $O(n)$ $\square$

Dynamic Programming!

Finding the Solution

Algorithm finds *value* of optimal solution: what if we want to find the solution itself?

Finding the Solution

Algorithm finds *value* of optimal solution: what if we want to find the solution itself?

- Idea 1: keep track of solution in another table (or in $\boldsymbol{M}$)

Finding the Solution

Algorithm finds *value* of optimal solution: what if we want to find the solution itself?

- Idea 1: keep track of solution in another table (or in M)
  - Uses lots of extra space, need to be careful about how much time spend copying/moving solutions

Finding the Solution

Algorithm finds *value* of optimal solution: what if we want to find the solution itself?

- Idea 1: keep track of solution in another table (or in $\boldsymbol{M}$)
  - Uses lots of extra space, need to be careful about how much time spend copying/moving solutions
- Better idea: Backtrack through completed table!

$OPT(p(n)) + v_n = M[p(n)] + v_n$

$OPT(n-1) = M[n-1]$

$M(n)$

Finding the Solution

Algorithm finds *value* of optimal solution: what if we want to find the solution itself?

- Idea 1: keep track of solution in another table (or in M)
  - Uses lots of extra space, need to be careful about how much time spend copying/moving solutions
- Better idea: Backtrack through completed table!

```
Solution(j) {
    If j = 0 then return ∅;
    else if v_j + M[p(j)] > M[j − 1] return {j} ∪ Solution(p(j));
    else return Solution(j − 1);
}
```

Finding the Solution

Algorithm finds *value* of optimal solution: what if we want to find the solution itself?

- Idea 1: keep track of solution in another table (or in M)
  - Uses lots of extra space, need to be careful about how much time spend copying/moving solutions
- Better idea: Backtrack through completed table!

```
Solution(j) {
    If j = 0 then return ∅;
    else if v_j + M[p(j)] > M[j − 1] return {j} ∪ Solution(p(j));
    else return Solution(j − 1);
}
```

Correctness: Direct from correctness of previous algorithm

Running Time: $O(n)$

Memoization vs Iteration: Top-Down vs Bottom-Up

Previous technique: "Memoization", "Top-Down Dynamic Programming"

- Remember outcome of recursive calls
- Starts at "top" problem, works way "down" via recursion

Memoization vs Iteration: Top-Down vs Bottom-Up

Previous technique: "Memoization", "Top-Down Dynamic Programming"

- Remember outcome of recursive calls
- Starts at "top" problem, works way "down" via recursion

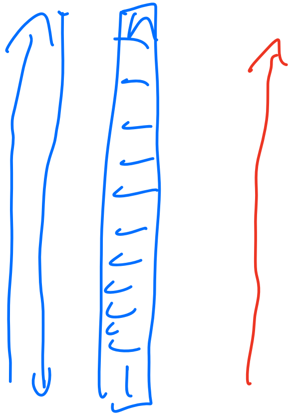
Alternative: "Bottom-Up Dynamic Programming"

- Start at "bottom" of table, work way up
- Every table entry we need already filled in!

Memoization vs Iteration: Top-Down vs Bottom-Up

Previous technique: "Memoization", "Top-Down Dynamic Programming"

- Remember outcome of recursive calls
- Starts at "top" problem, works way "down" via recursion



Alternative: "Bottom-Up Dynamic Programming"

- Start at "bottom" of table, work way up
- Every table entry we need already filled in!

Schedule {
  M[0] = 0;
  for(i = 1 to $\boldsymbol{n}$) {
    M[i] = max($\boldsymbol{v_i + M[p(i)]}$, $\boldsymbol{M[i-1]}$);
  }
  return M[n];
}

Top-Down vs Bottom-Up (cont'd)

Some people only call bottom-up dynamic programming, but this is ridiculous

Top-Down vs Bottom-Up (cont'd)

Some people only call bottom-up dynamic programming, but this is ridiculous

Top-Down pros:

- If $M[j]$ doesn't need to be computed (doesn't appear in recursion tree for $M[n]$), won't waste time on it!
- Algorithm design relatively easy: write recursive algorithm, remember (memoize) answers

Top-Down vs Bottom-Up (cont'd)

Some people only call bottom-up dynamic programming, but this is ridiculous

Top-Down pros:

- If $M[j]$ doesn't need to be computed (doesn't appear in recursion tree for $M[n]$), won't waste time on it!
- Algorithm design relatively easy: write recursive algorithm, remember (memoize) answers

Bottom-up pros:

- Easier to analyze running time: sum over all table entries of time to compute entry
- Often faster in practice (iteration vs recursion)

Top-Down vs Bottom-Up (cont'd)

Some people only call bottom-up dynamic programming, but this is ridiculous

Top-Down pros:

- If $\boldsymbol{M}[\boldsymbol{j}]$ doesn't need to be computed (doesn't appear in recursion tree for $\boldsymbol{M}[\boldsymbol{n}]$), won't waste time on it!
- Algorithm design relatively easy: write recursive algorithm, remember (memoize) answers

Bottom-up pros:

- Easier to analyze running time: sum over all table entries of time to compute entry
- Often faster in practice (iteration vs recursion)

Use whatever you feel more comfortable with (most experienced people use bottom-up)

Principles of Dynamic Programming (CLRS 15.3)

Main step: break problem into subproblems

- WIS: Subproblems $\{1, \ldots, i\}$ (prefixes)
- Often determined by choice ("is n in S^*?")
- Want small (polynomial) number of subproblems (table entries)

Prove *optimal substructure*: Optimal solution to subproblem can be found from optimal solutions to *smaller* subproblems

- Not an algorithmic statement! *Smaller* very important!

Turn optimal substructure theorem into algorithm (top-down or bottom-up) which fills in table indexed by subproblems

- Correctness: induction and optimal substructure theorem
- Running time: sum of time of all table entries
  - Often (not always) just (# table entries) × (time per entry)