Advanced Algorithmic Techniques (COMP523)

Greedy Algorithms 3

Recap and plan

- **Last 3 lectures:**
  - Greedy Algorithms
    - Interval Scheduling, Minimum Spanning Tree, Max-Spacing Clustering
- **This lecture:**
  - Dynamic Programming
  - Weighted Interval Scheduling

Dynamic Programming

- An technique for solving optimisation problems.
- Term attributed to Bellman (1950s).
    - “Programming” as in “Planning” or “Optimising”.

Dynamic Programming

Dynamic Programming

- The paradigm of dynamic programming:

Dynamic Programming

- The paradigm of dynamic programming:
  - Given a problem P, define a sequence of subproblems, with the following properties:

Dynamic Programming

- The paradigm of dynamic programming:
  - Given a problem P, define a sequence of subproblems, with the following properties:
    - The subproblems are ordered from the smallest to the largest.

Dynamic Programming

- The paradigm of dynamic programming:
  - Given a problem P, define a sequence of subproblems, with the following properties:
    - The subproblems are ordered from the smallest to the largest.
    - The largest problem is our original problem P.

Dynamic Programming

- The paradigm of dynamic programming:
  - Given a problem P, define a sequence of subproblems, with the following properties:
    - The subproblems are ordered from the smallest to the largest.
    - The largest problem is our original problem P.
    - The optimal solution of a subproblem can be constructed from the optimal solutions of sub-sub-problems. (*Optimal Substructure*).

Dynamic Programming

- The paradigm of dynamic programming:
  - Given a problem P, define a sequence of subproblems, with the following properties:
    - The subproblems are ordered from the smallest to the largest.
    - The largest problem is our original problem P.
    - The optimal solution of a subproblem can be constructed from the optimal solutions of sub-sub-problems. (*Optimal Substructure*).
  - Solve the subproblems from the smallest to the largest. When you solve a subproblem, store the solution (e.g., in an array) and use it to solve the larger subproblems.

Recall: Interval Scheduling

- A set of requests $\{1, 2, \ldots, n\}$.
  - Each request has a starting time $s(i)$ and a finishing time $f(i)$.
  - Alternative view: Every request is an interval $[s(i), f(i)]$.
- Two requests i and j are compatible if their respective intervals do not overlap.
- Goal: Output a schedule which maximises the number of compatible intervals.

Weighted Interval Scheduling

- A set of requests $\{1, 2, \ldots, n\}$.
  - Each request has a starting time $s(i)$, a finishing time $f(i)$, and a value $v(i)$.
  - Alternative view: Every request is an interval $[s(i), f(i)]$ associated with a value $v(i)$.
- Two requests i and j are compatible if their respective intervals do not overlap.
- Goal: Output a schedule which maximises the total value of compatible intervals.

Greedy Approaches

- Which one of the following Greedy Algorithms might have a chance to work?
  - Earliest starting time.
  - Smallest interval.
  - Minimum number of conflicts.
  - Earliest finishing time.

Greedy Approaches

- Which one of the following Greedy Algorithms might have a chance to work?
  - Earliest starting time.
  - Smallest interval.
  - Minimum number of conflicts.
  - Earliest finishing time.

Does it work?

Does it work?

value=1

Does it work?

Does it work?

Does it work?

Does it work?

Does it work?

No approach that ignores the values can work!

Greedy Approaches

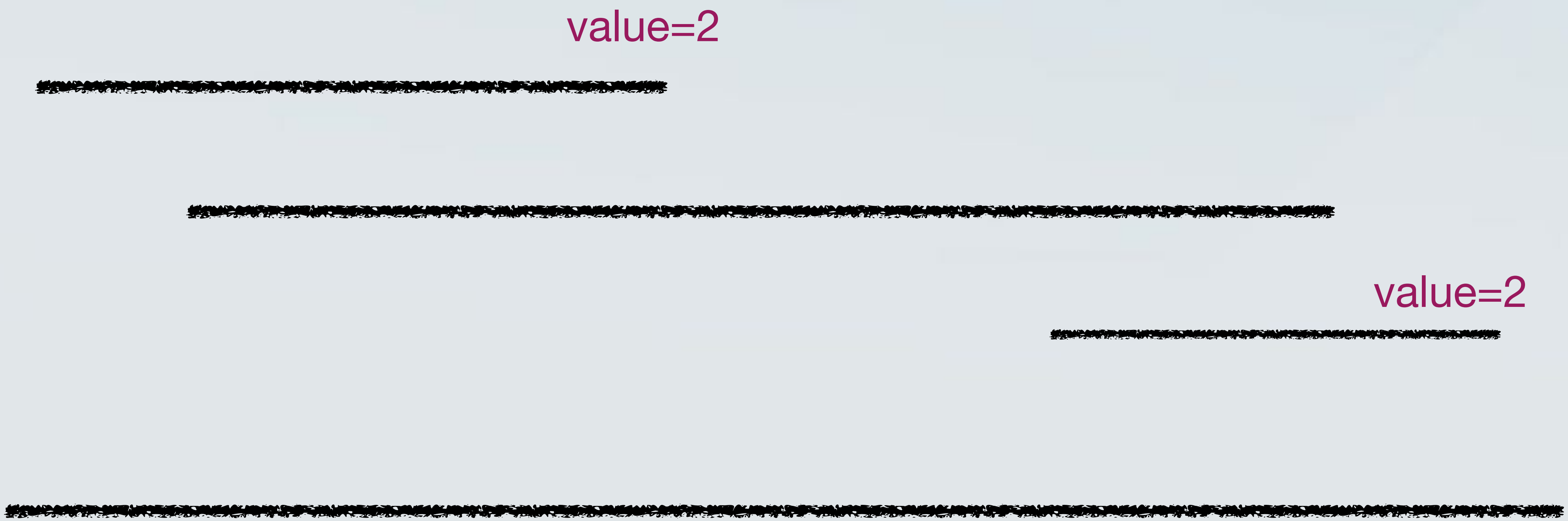
- Which one of the following Greedy Algorithms might have a chance to work?
  - Earliest starting time.
  - Smallest interval.
  - Minimum number of conflicts.
  - Earliest finishing time.
  - Largest value.

Does it work?

Does it work?

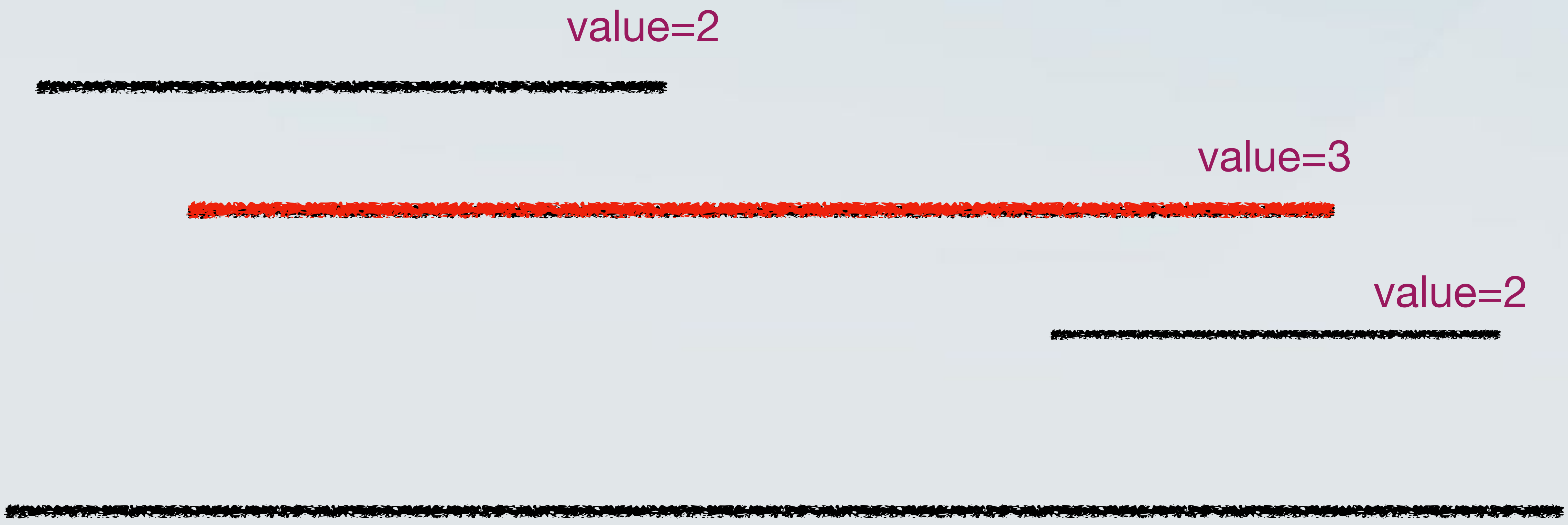
value=2

Does it work?



Does it work?

Does it work?



A view of the input

- Consider the intervals in sorted order of non-decreasing finishing time, i.e., $f(1) \leq f(2) \leq \ldots \leq f(n)$.

- For an interval $j = (s(j), f(j))$, let p_j be the largest index $i < j$ such that intervals i and j are disjoint.

  - i.e., i is the first interval in the ordering that ends before j begins.

  - if no such interval exists, define $p_j = 0$.

Example

$v(1)=2,\ p_1 = 0$

$v(2)=4,\ p_2 = 0$

$v(3)=4,\ p_3 = 1$

$v(4)=7,\ p_4 = 0$

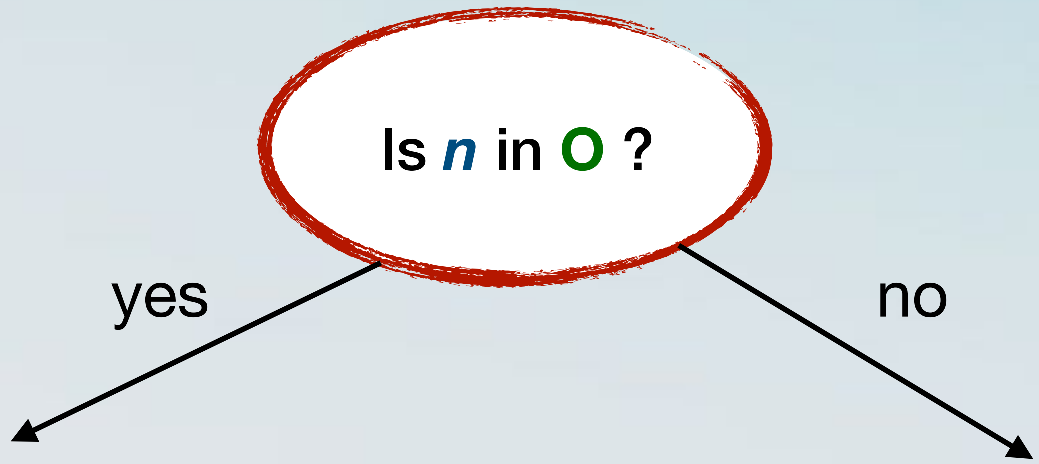
$v(5)=2,\ p_5 = 3$

$v(6)=1,\ p_6 = 3$

Step-by-step?

- Let O be the optimal schedule.
- Fact: O either contains interval *n* or not.

Building up a solution



If n is in O

If *n* is in O

- What does that mean for the other intervals?

If *n* is in O

- What does that mean for the other intervals?
- Any interval that overlaps with *n* cannot be in O.

If *n* is in O

- What does that mean for the other intervals?
- Any interval that overlaps with *n* cannot be in O.
- Any interval *j* > p_n cannot be in O.

If n is in O

- What does that mean for the other intervals?
- Any interval that overlaps with n cannot be in O.
- Any interval $j > p_n$ cannot be in O.
- O contains an optimal solution O' of the subproblem $\{1, 2, \ldots, p_n\}$ (why?)

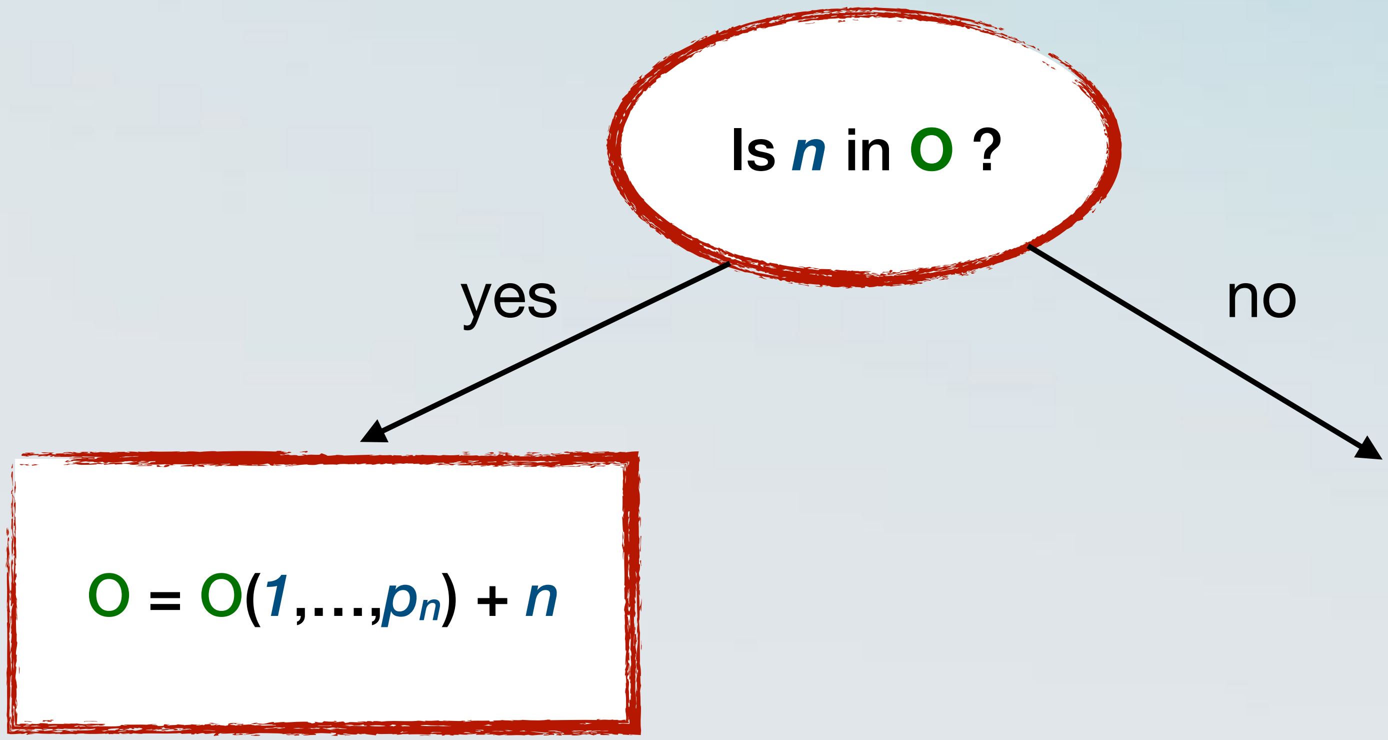
If *n* is in O

- What does that mean for the other intervals?
- Any interval that overlaps with *n* cannot be in O.
- Any interval $j > p_n$ cannot be in O.
- O contains an optimal solution O' of the subproblem $\{1, 2, \ldots, p_n\}$ (why?)
  - Because otherwise we could replace O with $O' \cup \{n\}$ and obtain a better solution.

If n is in O

- What does that mean for the other intervals?
- Any interval that overlaps with n cannot be in O.
- Any interval $j > p_n$ cannot be in O.
- O contains an optimal solution O' of the subproblem $\{1, 2, \ldots, p_n\}$ (why?)
    - Because otherwise we could replace O with O' $\cup \{n\}$ and obtain a better solution.
- Lets use O(i, …, j) to denote the optimal solution on (sorted) intervals i, … , j.

Building up a solution



If *n* is not in O

If n is not in O

- Then O = O(1, …,n-1)

If n is not in O

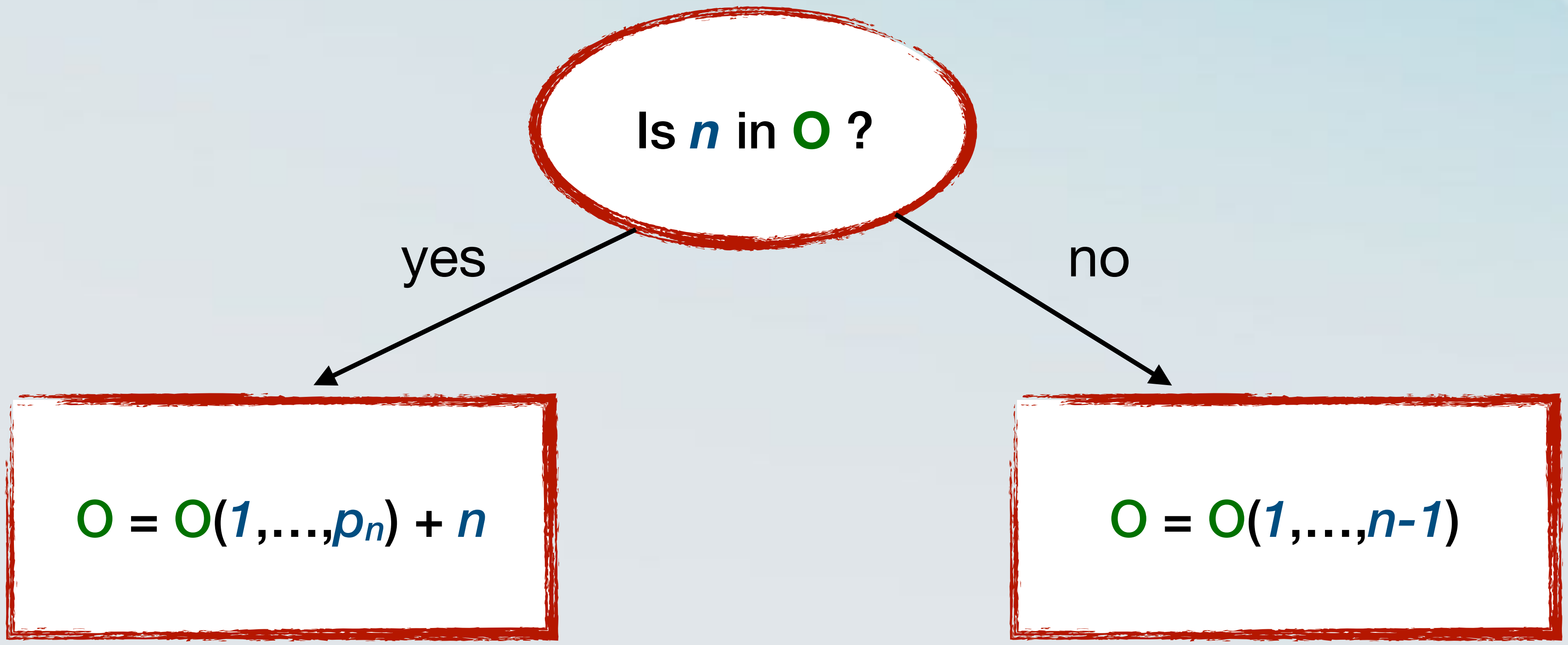
- Then O = O($1, \ldots, n-1$)

- Same argument: Since n is not chosen, all intervals $1, \ldots, n-1$ are “free” to be chosen.

If *n* is not in O

- Then O = O(1, …,n-1)
- Same argument: Since *n* is not chosen, all intervals 1, …, n-1 are “free” to be chosen.
- Not picking the optimal schedule for them would violate the optimality of O.

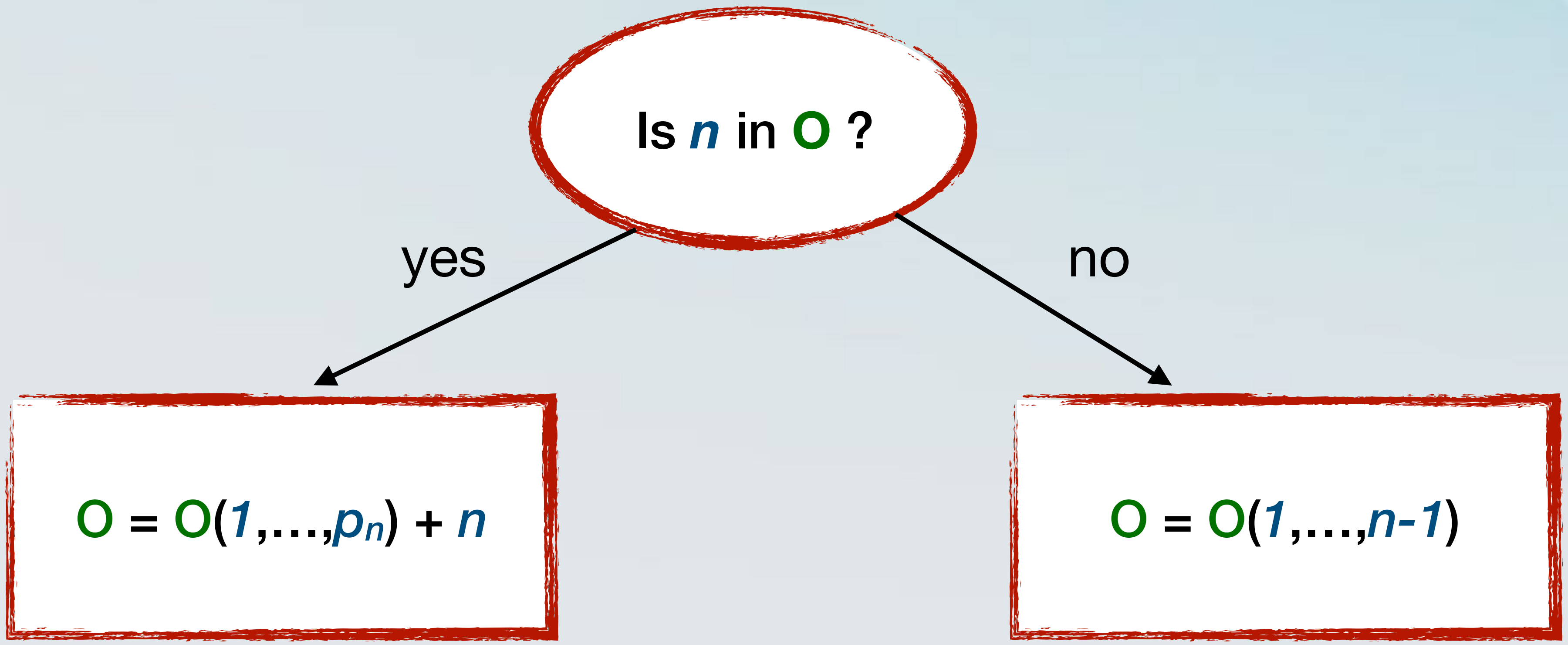
Building up a solution



Building up a solution

- So, in order to find O, it suffices to look at smaller problems and find O(1, … , j) for some j.
- Let O_j be a shorthand for O(1, … , j) and let OPT(j) be its total value.
- Define OPT(0) = 0.
- Then, $O = O_n$ with value OPT(n).

Building up a solution



Generalising

Is j in **O** ?

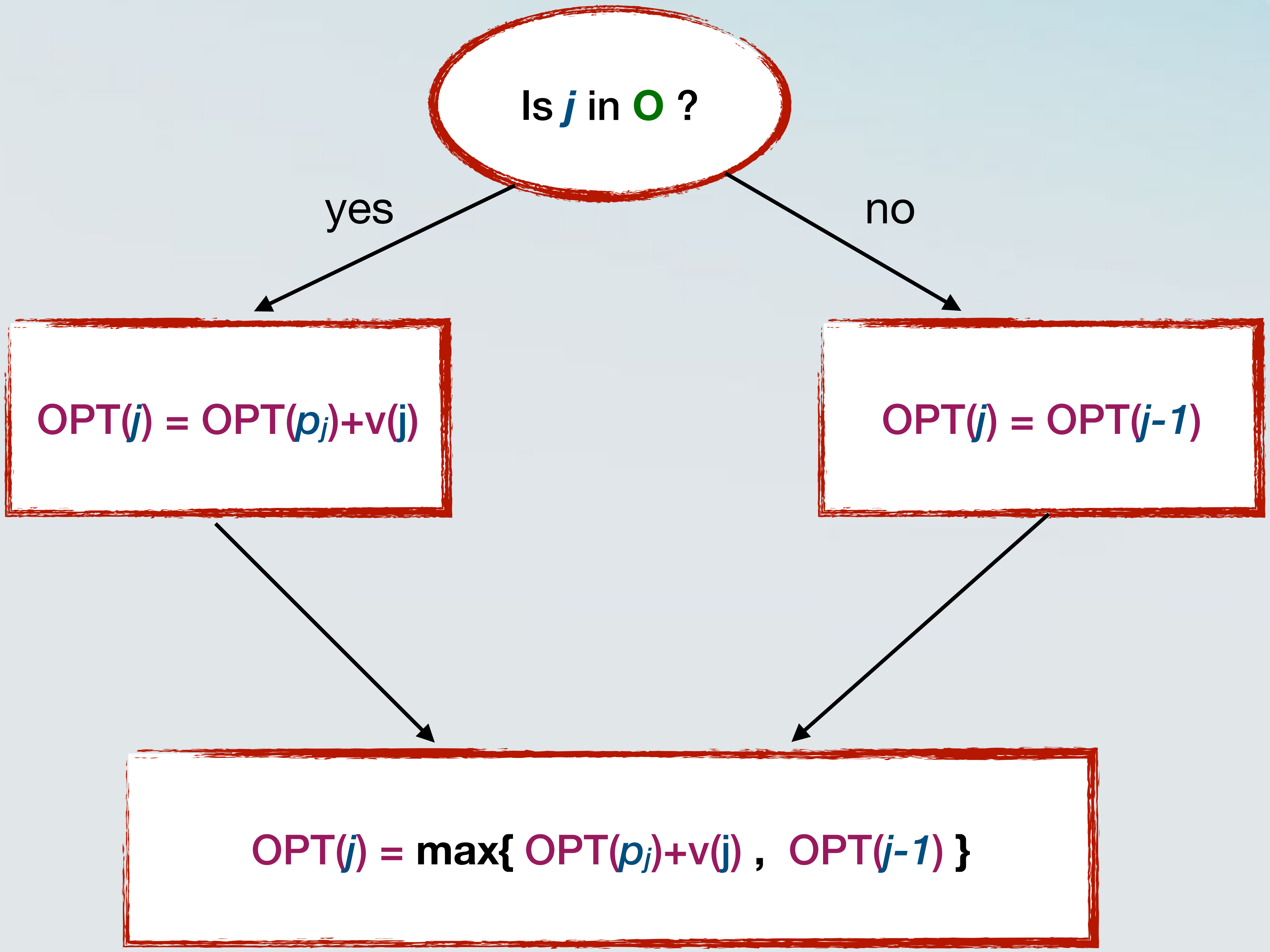
yes

no

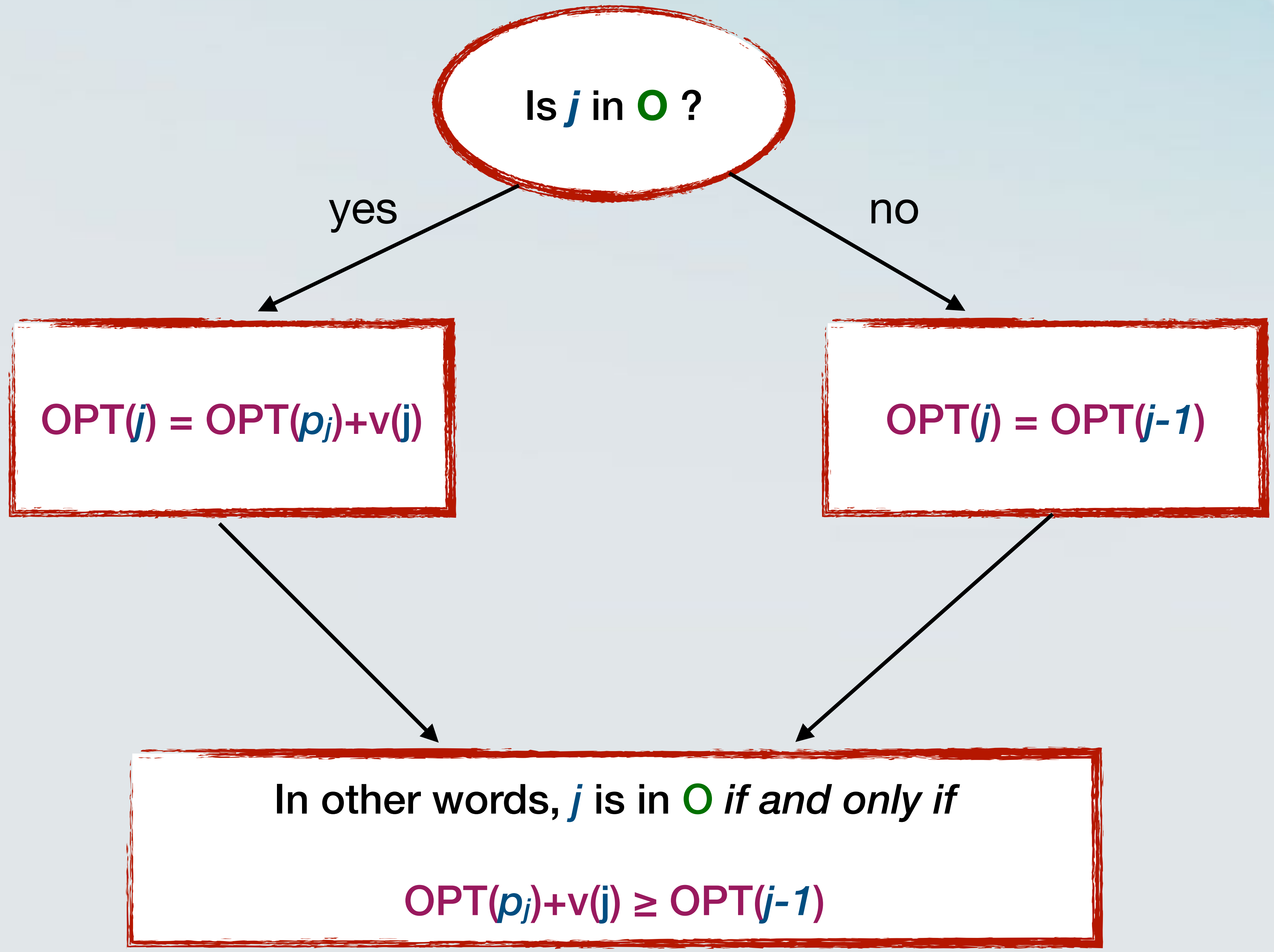
$\mathrm{OPT}(j) = \mathrm{OPT}(p_j)+v(j)$

$\mathrm{OPT}(j) = \mathrm{OPT}(j-1)$

Generalising



Generalising



Building up a solution

$$OPT(j) = \max\{ OPT(p_j)+v(j) \;,\; OPT(j-1) \}$$

Building up a solution

$$\boxed{OPT(j) = \max\{ OPT(p_j)+v(j)\ ,\ \ OPT(j-1) \}}$$

- What does this look like?

Building up a solution

$$\boxed{OPT(j) = \max\{ OPT(p_j)+v(j)\ ,\ OPT(j-1) \}}$$

- What does this look like?
- Assume that there was an algorithm that inputed $\{1, \ldots, j\}$ and outputted $OPT(j)$.

Building up a solution

$$\boxed{OPT(j) = \max\{\ OPT(p_j)+v(j)\ ,\ OPT(j-1)\ \}}$$

- What does this look like?
- Assume that there was an algorithm that inputed $\{1, \ldots, j\}$ and outputted $OPT(j)$.
- It's a recurrence relation!

Building up a solution

- What does this look like?

- Assume that there was an algorithm that inputed $\{1, \ldots, j\}$ and outputted $\mathrm{OPT}(j)$.

- It’s a recurrence relation!

Building up a solution

- What does this look like?
- Assume that there was an algorithm that inputed {*1*, …, *j*} and outputted OPT(*j*).
- It's a recurrence relation!

```
ComputeOpt(j)
    If j = 0 then
        Return 0
    Else
        Return max{v(j) + ComputeOpt(p_j) , ComputeOpt(j-1)}
    EndIf
```

Correctness

- **ComputeOPT**(j) correctly computes OPT(j) for each j=1, …n

Correctness

- **ComputeOPT**(j) correctly computes OPT(j) for each j=1, …n
- Proof by induction:

Correctness

- ComputeOPT(j) correctly computes OPT(j) for each j=1, …n
- Proof by induction:
  - Base Case: OPT(0) = 0 by definition.

Correctness

- ComputeOPT(j) correctly computes OPT(j) for each j=1, …n

- Proof by induction:

  - Base Case: OPT(0) = 0 by definition.

  - Inductive step: Assume that it is true for all $i < j$. (inductive hypothesis).

    Return **max{**v(j) + ComputeOpt(p_j) , ComputeOpt(j-1)**}**

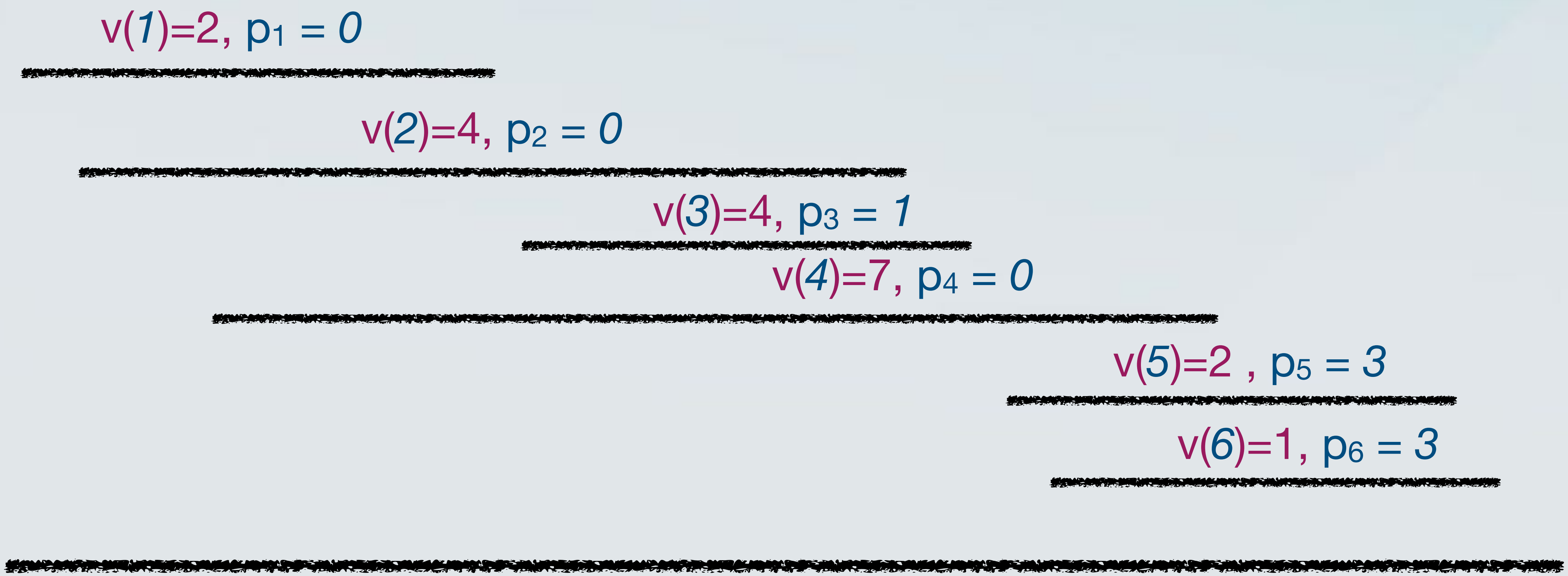
Correctness

- **ComputeOPT**(j) correctly computes OPT(j) for each j=1, …n
- Proof by induction:
  - Base Case: OPT(0) = 0 by definition.
  - Inductive step: Assume that it is true for all $i < j$. (inductive hypothesis).

    Return **max{**v(j) + **ComputeOpt**(p_j) **, ComputeOpt**(j-1)**}**

$$\boxed{OPT(j) = \max\{\, OPT(p_j)+v(j)\ ,\ OPT(j-1)\,\}}$$

Example



Example

OPT(6)

$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

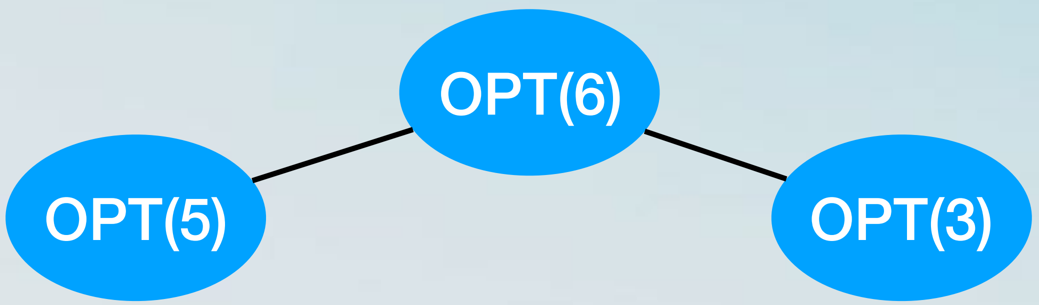
$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Return **max{**v(j) + **ComputeOpt**(p_j) **, ComputeOpt**(j-1)**}**

Example



$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

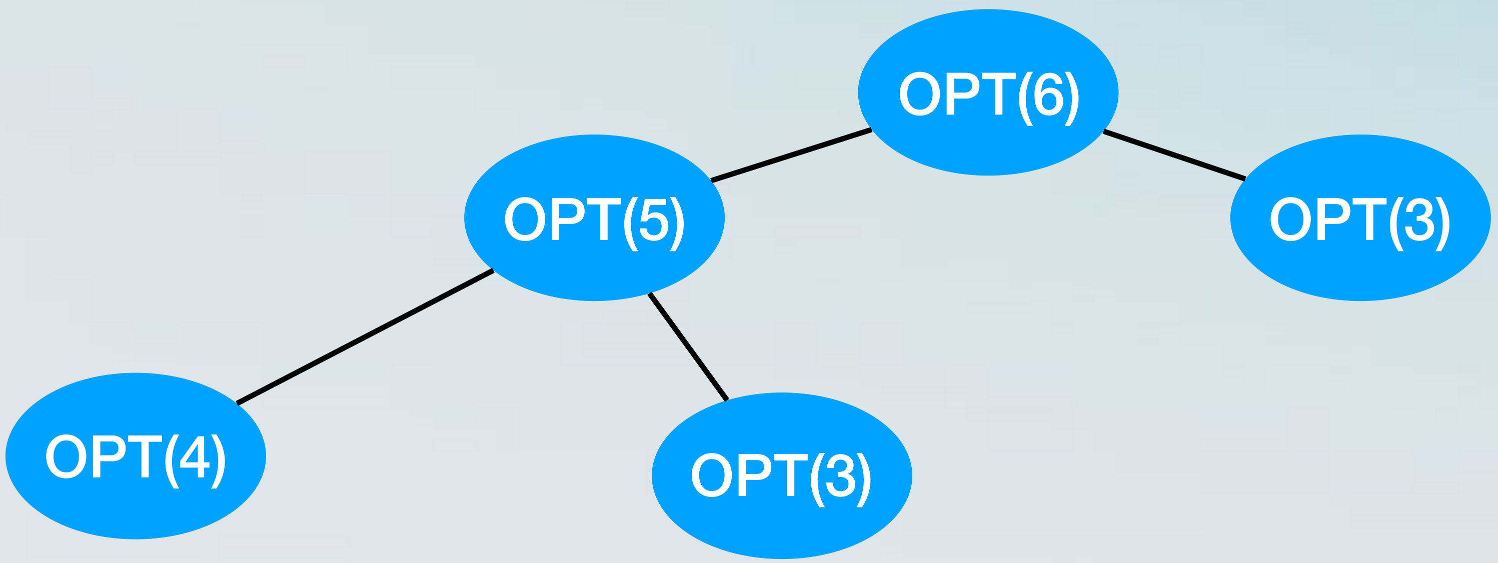
$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Return **max{**$v(j)$ + **ComputeOpt**(p_j) , **ComputeOpt**($j-1$)**}**

Example



$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

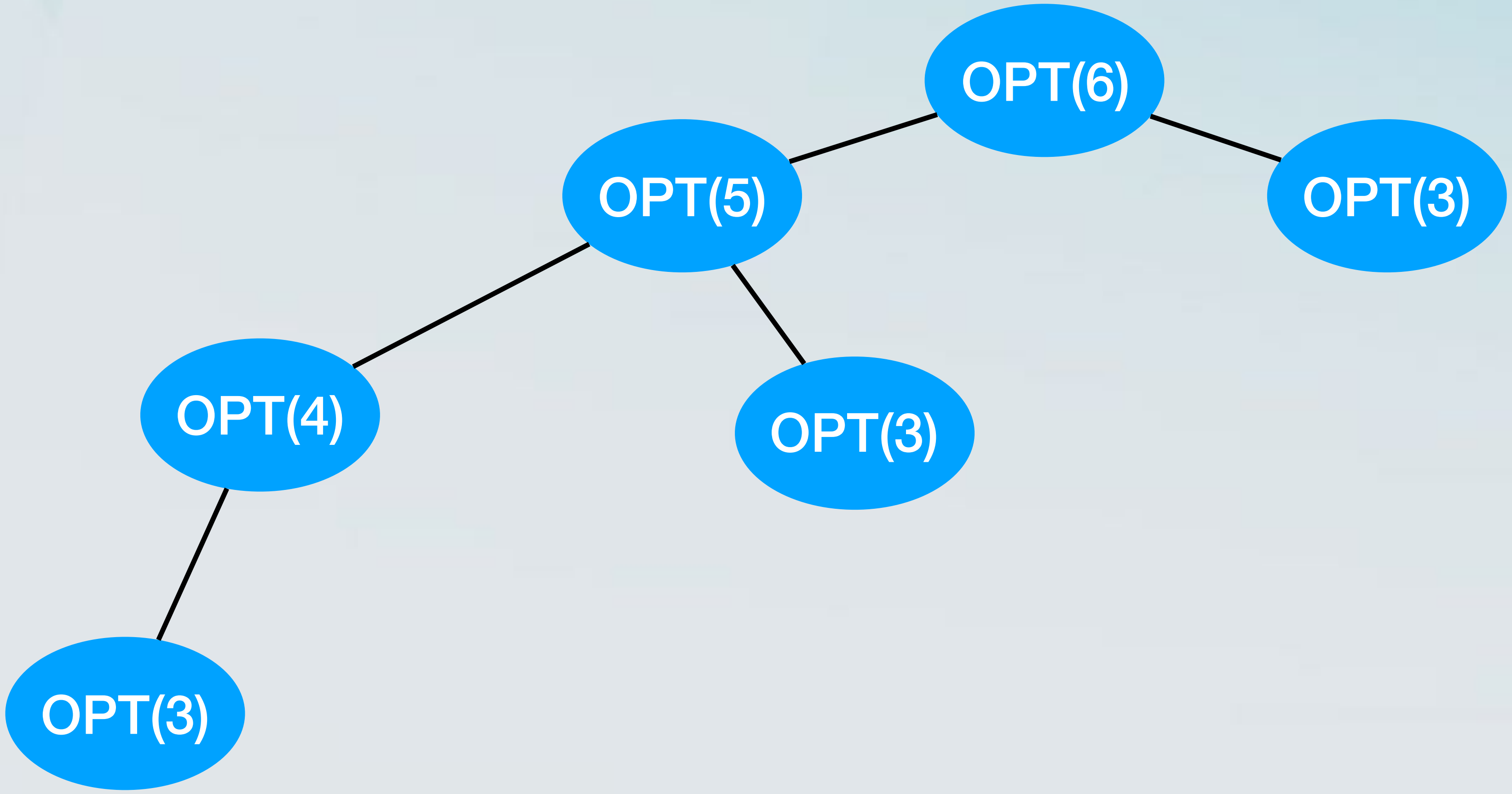
$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Return **max**{$v(j)$ + ComputeOpt(p_j) , ComputeOpt(j-1)}

Example



$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

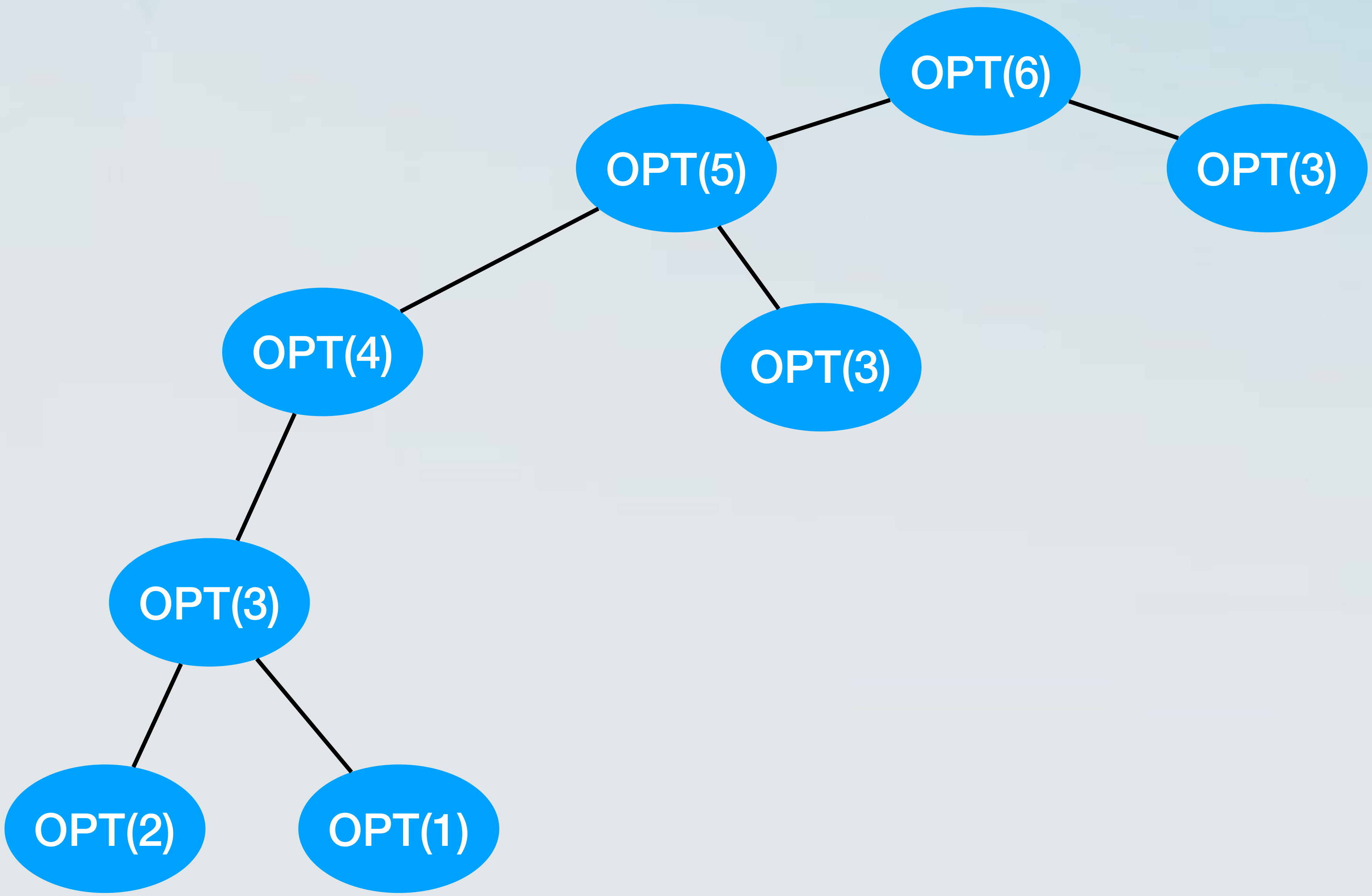
$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Return **max{**$v(j)$ + **ComputeOpt**(p_j) , **ComputeOpt**($j-1$)**}**

Example



$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

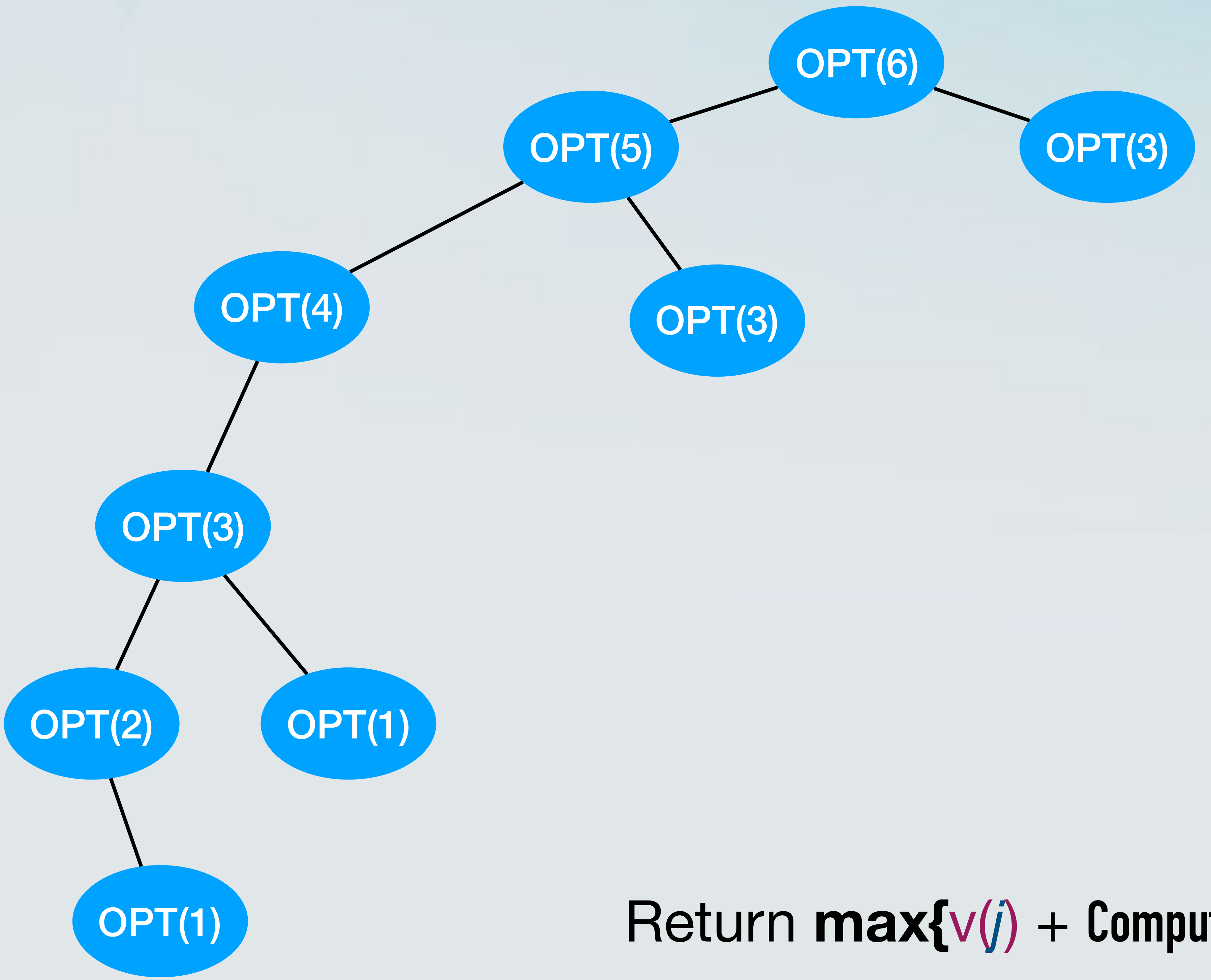
$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Return **max{**$v(j)$ + **ComputeOpt**(p_j) **, ComputeOpt**($j-1$)**}**

Example



$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

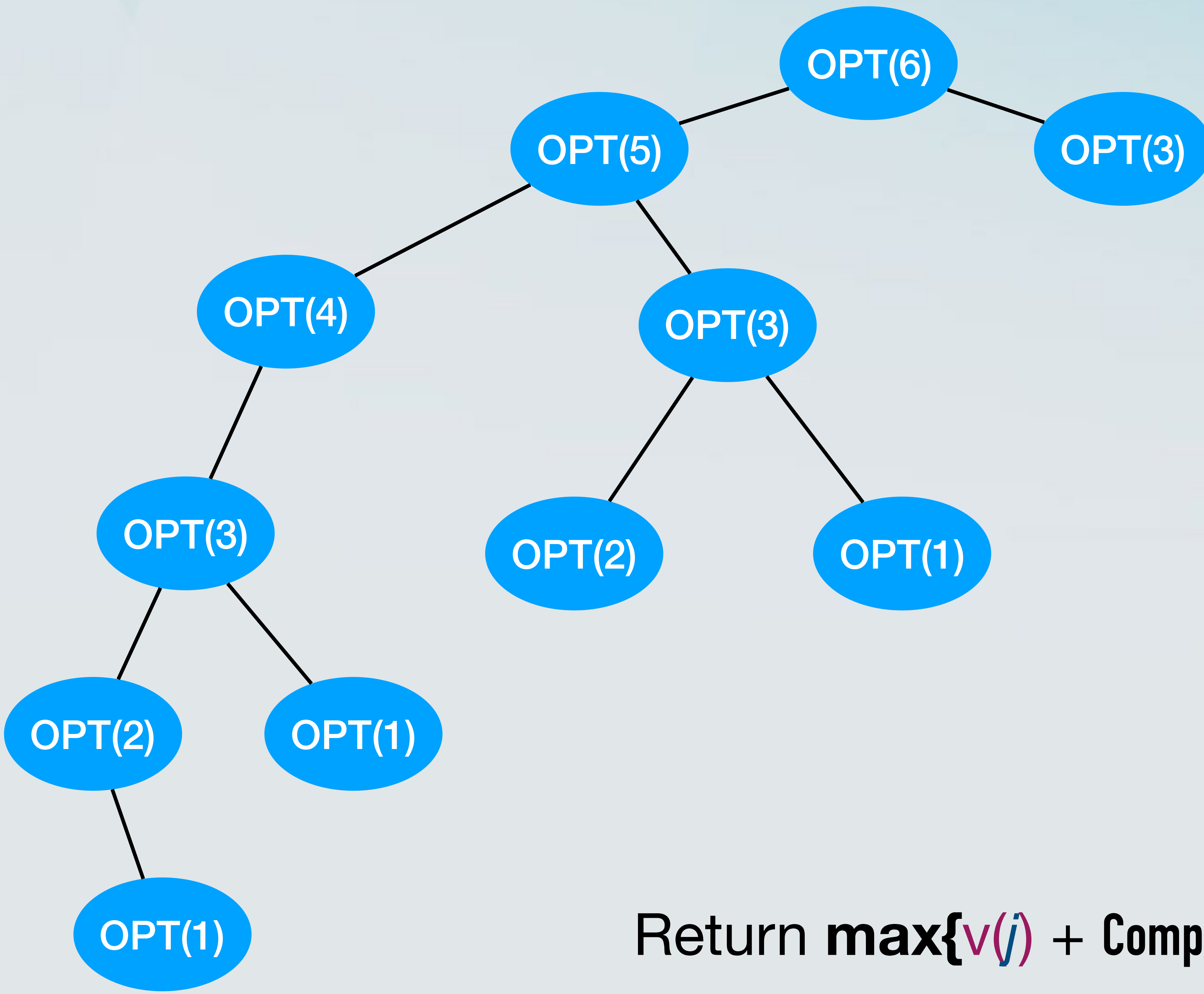
$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Return **max{**$v(j)$ + **ComputeOpt(**p_j**)** , **ComputeOpt(**$j-1$**)}**

Example



$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

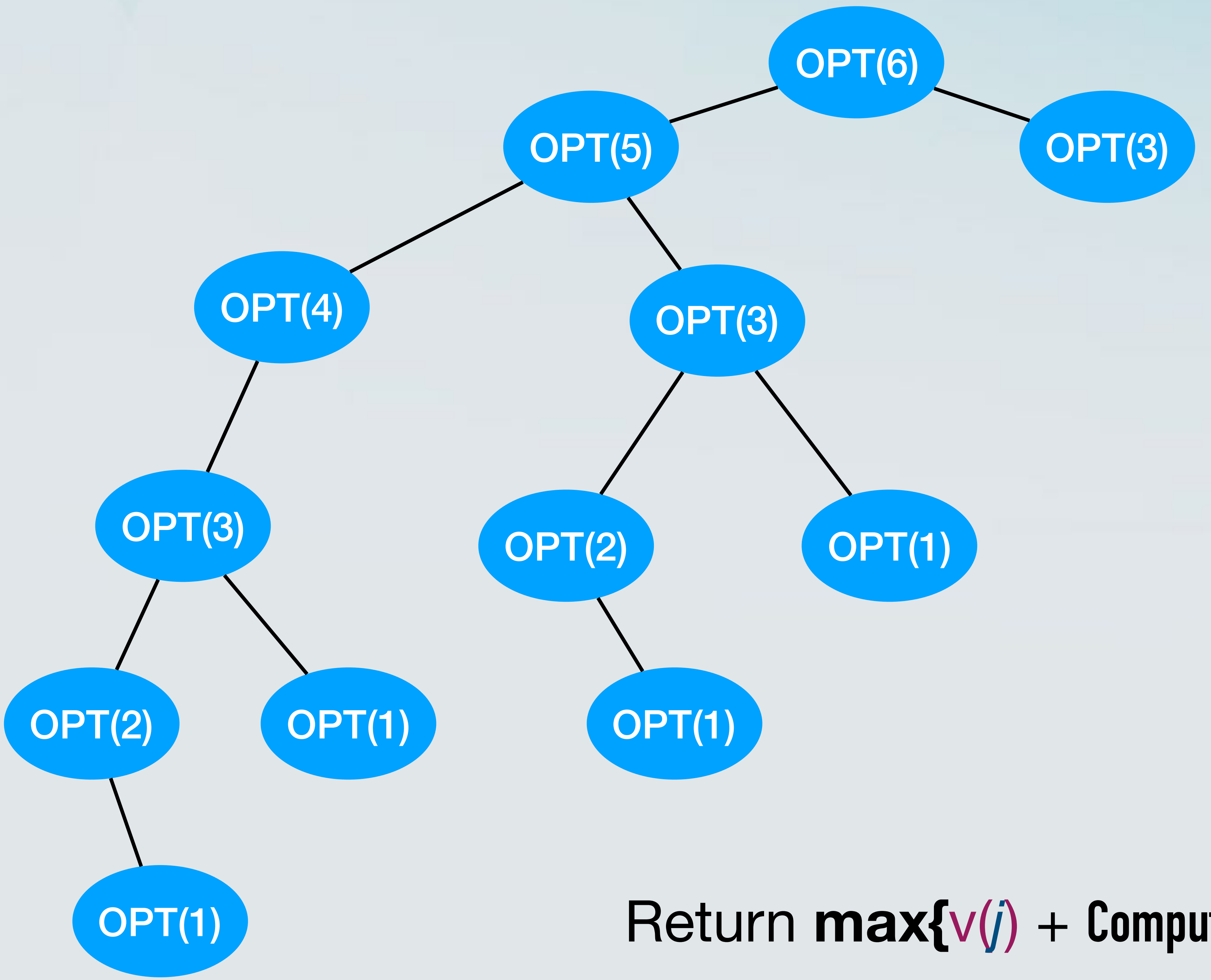
$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Return **max{**$v(j)$ + **ComputeOpt**(p_j) **, ComputeOpt**($j-1$)**}**

Example



$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

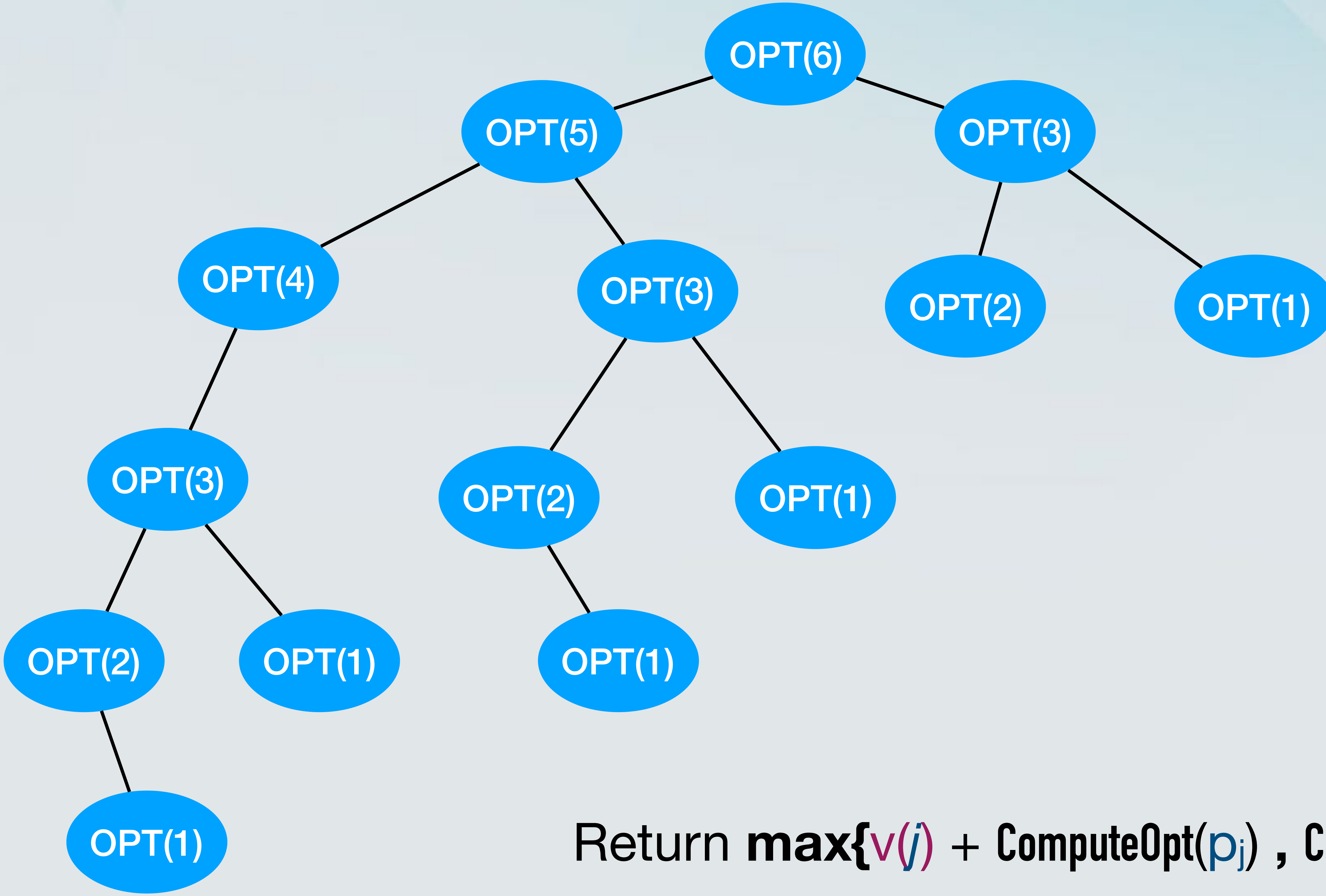
$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Return **max{**v(*j*) + **ComputeOpt**(p_j) , **ComputeOpt**(*j*-*1*)**}**

Example



$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

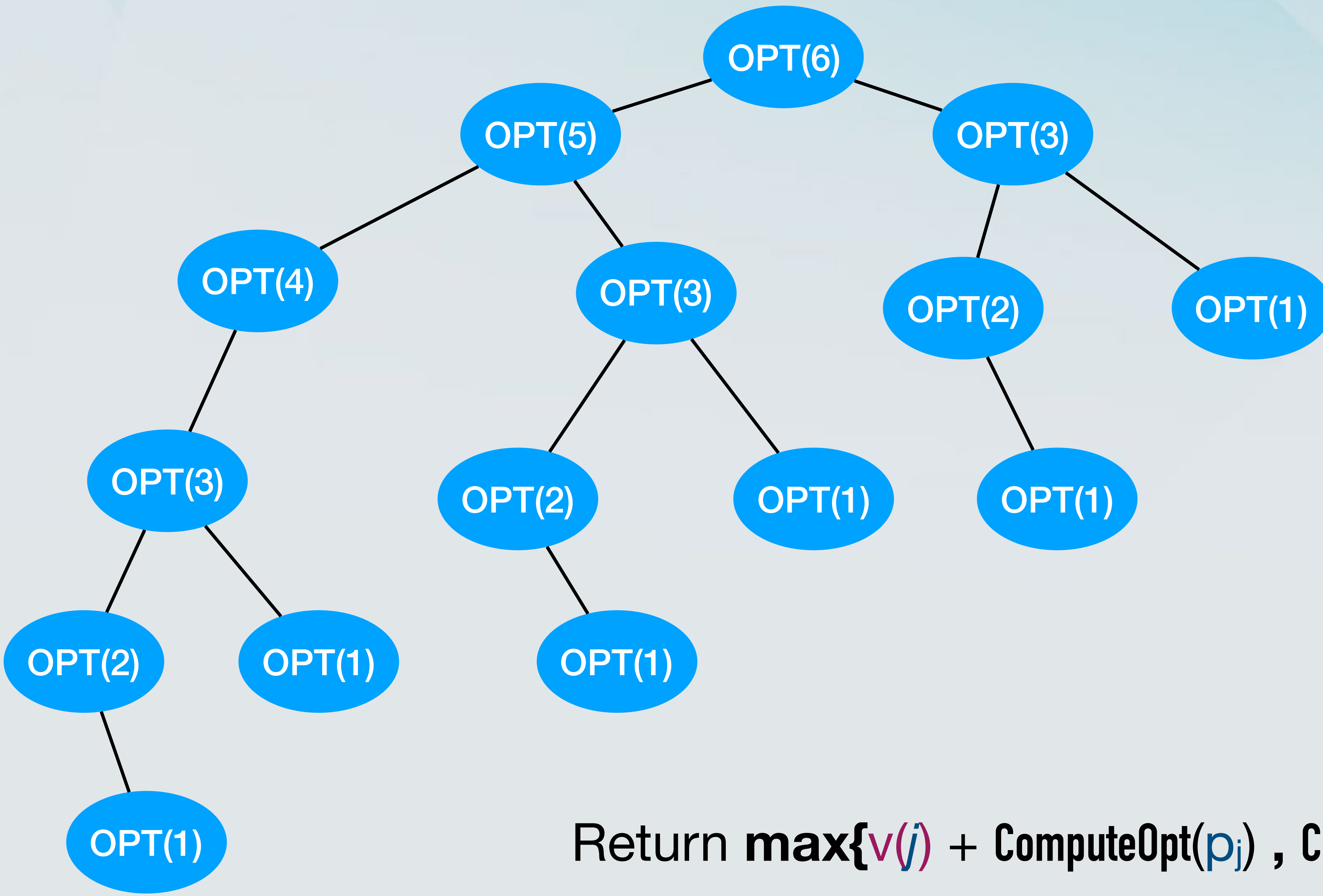
$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Return **max**{v(j) + ComputeOpt(p_j) , ComputeOpt($j-1$)}

Example



$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Return **max{**$v(j)$ + **ComputeOpt**(p_j) **, ComputeOpt**($j-1$)**}**

Another example

$v(1)=1$, $p_1 = 0$

$v(2)=1$, $p_2 = 0$

$v(3)=1$, $p_3 = 1$

$v(4)=1$, $p_4 = 2$

$v(5)=1$, $p_5 = 3$

$v(6)=1$, $p_6 = 4$

Another example

$v(1)=1$, $p_1 = 0$

$v(2)=1$, $p_2 = 0$

$v(3)=1$, $p_3 = 1$

$v(4)=1$, $p_4 = 2$

$v(5)=1$, $p_5 = 3$

$v(6)=1$, $p_6 = 4$

ComputeOpt(6) requires **ComputeOpt**(5) and **ComputeOpt**(4)

Another example

$v(1)=1$, $p_1 = 0$

$v(2)=1$, $p_2 = 0$

$v(3)=1$, $p_3 = 1$

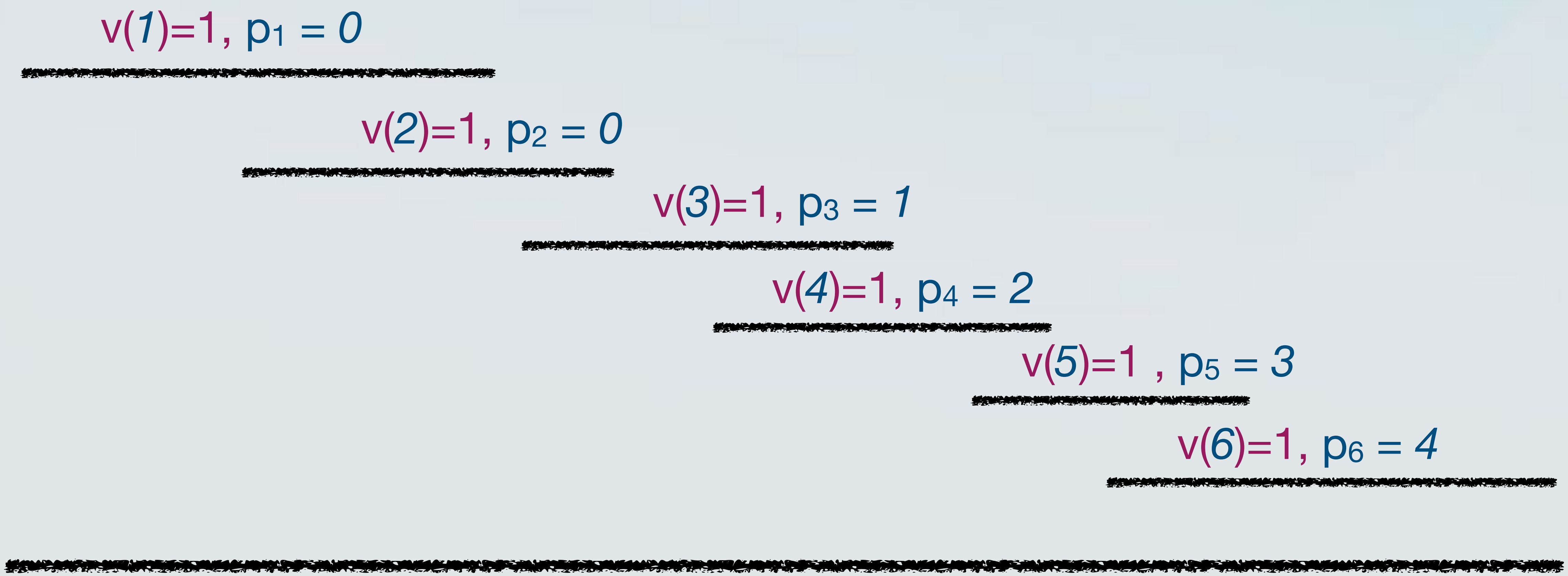
$v(4)=1$, $p_4 = 2$

$v(5)=1$, $p_5 = 3$

$v(6)=1$, $p_6 = 4$

ComputeOpt(6) requires **ComputeOpt**(5) and **ComputeOpt**(4)
ComputeOpt(5) requires **ComputeOpt**(4) and **ComputeOpt**(3)

Another example



ComputeOpt(6) requires **ComputeOpt**(5) and **ComputeOpt**(4)
ComputeOpt(5) requires **ComputeOpt**(4) and **ComputeOpt**(3)
ComputeOpt(4) requires **ComputeOpt**(3) and **ComputeOpt**(2)

Running time

Running time

- What is the running time of the algorithm?

Running time

- What is the running time of the algorithm?
- A problem of size *j* requires solving problems of sizes *j-1* and *j-2*.

Running time

- What is the running time of the algorithm?
- A problem of size *j* requires solving problems of sizes *j-1* and *j-2*.
- Do you know any numbers for which F(*n*) = F(*n-1*) + F(*n-2*) ?

Running time

- What is the running time of the algorithm?
- A problem of size j requires solving problems of sizes $j-1$ and $j-2$.
- Do you know any numbers for which $F(n) = F(n-1) + F(n-2)$?
  - Fibonacci numbers.

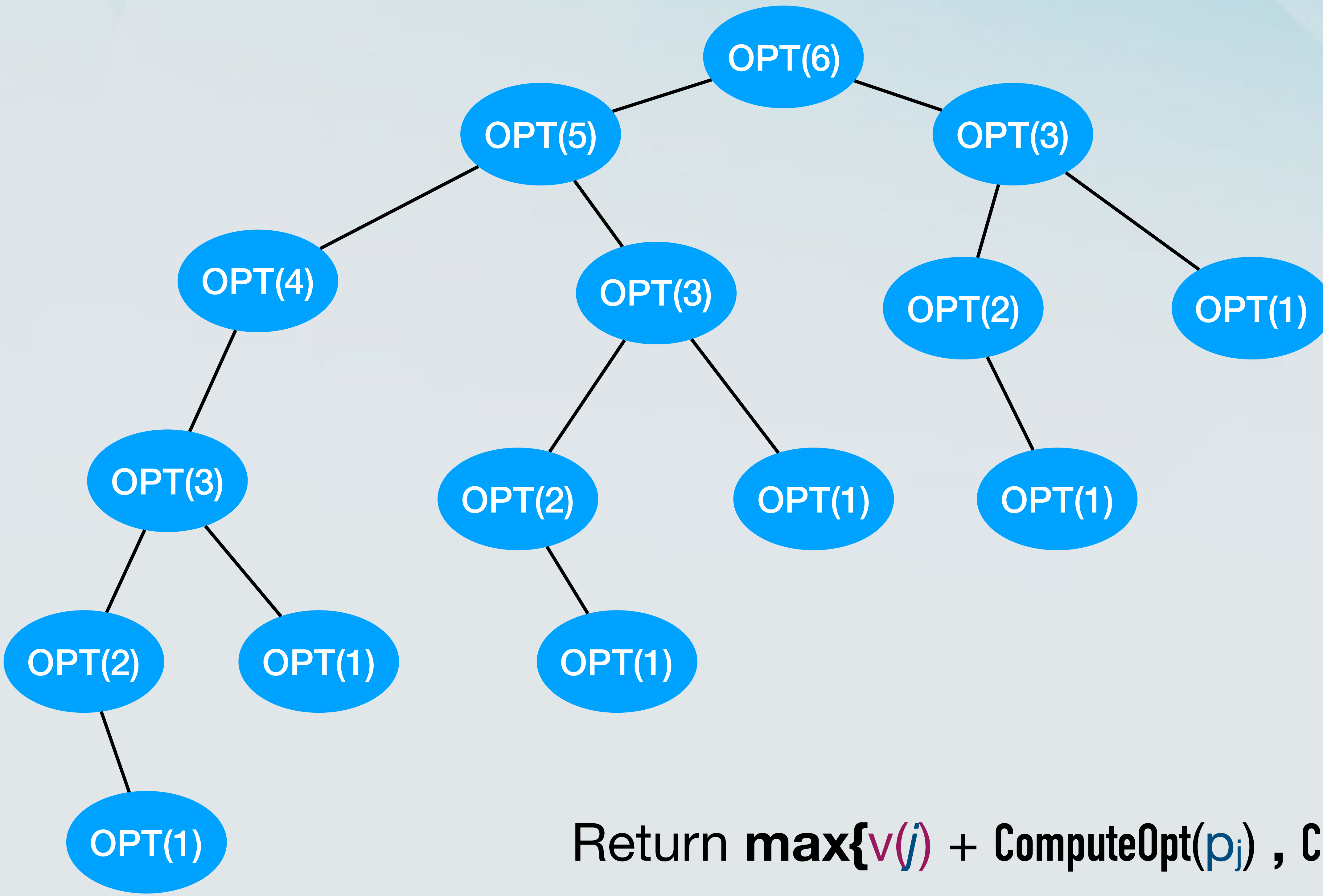
Running time

- What is the running time of the algorithm?
- A problem of size j requires solving problems of sizes $j-1$ and $j-2$.
- Do you know any numbers for which $F(n) = F(n-1) + F(n-2)$?
  - Fibonacci numbers.
- The nth Fibonacci number is approximately $\phi^n/\sqrt{5}$

Running time

- What is the running time of the algorithm?
- A problem of size j requires solving problems of sizes $j-1$ and $j-2$.
- Do you know any numbers for which $F(n) = F(n-1) + F(n-2)$?
  - Fibonacci numbers.
- The nth Fibonacci number is approximately $\phi^n/\sqrt{5}$
- The running time of our algorithm is $\Omega(2^n)$!

Are we being smart enough?



$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

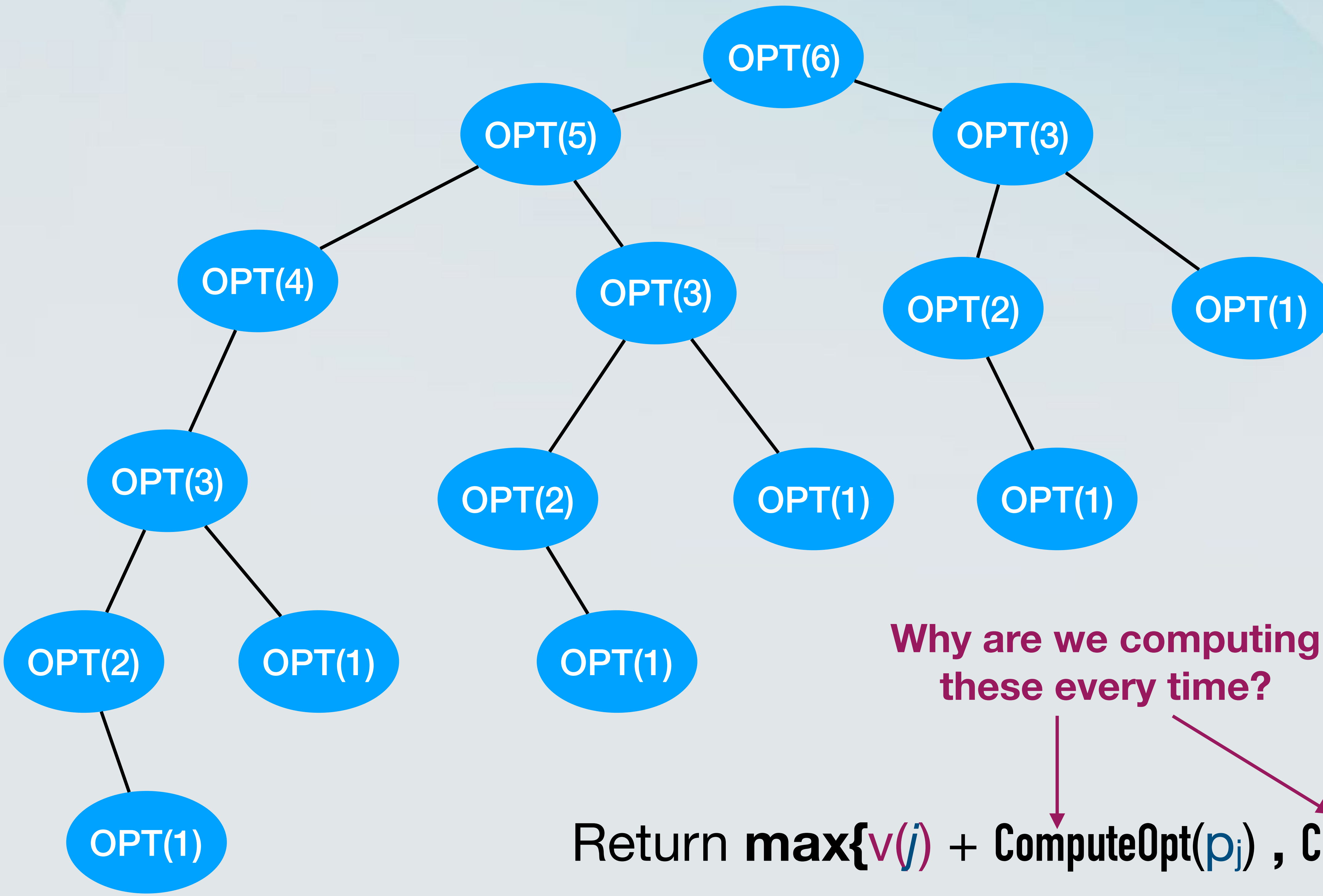
$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Return **max{**$v(j)$ + ComputeOpt(p_j) , ComputeOpt($j-1$)**}**

Are we being smart enough?



$p_6 = 3$

$p_5 = 3$

$p_4 = 0$

$p_3 = 1$

$p_2 = 0$

$p_1 = 0$

Why are we computing these every time?

Return **max**{$v(j)$ + ComputeOpt(p_j) , ComputeOpt($j-1$)}

Memoization

- Compute **ComputeOpt**(j) once for every j.
- Store it in an accessible place to use again in the future.
- Keep an array M[0, … ,n].
  - Initially M[j] = “empty” for all j.
  - When **ComputeOpt**(j) is calculated, M[j] = **ComputeOpt**(j)

A more clever implementation

M-ComputeOpt(j)

If j=0 then
Return 0

Else if M[j] is not empty then
Return M[j]

Else
M[j] = **max**{v(j) + M-ComputeOpt(p_j) , M-ComputeOpt($j-1$)}
Return M[j]

EndIf

Running time

Running time

- In each call of M-ComputeOpt, there is a constant number of operations, besides the recursive calls. So the running time is bounded by the number of recursive calls.

Running time

- In each call of M-ComputeOpt, there is a constant number of operations, besides the recursive calls. So the running time is bounded by the number of recursive calls.

- The two recursive call only happen when M[j] is empty.

Running time

- In each call of **M-ComputeOpt**, there is a constant number of operations, besides the recursive calls. So the running time is bounded by the number of recursive calls.

- The two recursive call only happen when M[*j*] is empty.

- But when they happens, M[*j*] is no longer empty.

Running time

- In each call of M-ComputeOpt, there is a constant number of operations, besides the recursive calls. So the running time is bounded by the number of recursive calls.
- The two recursive call only happen when M[j] is empty.
- But when they happens, M[j] is no longer empty.
- So the recursively calls only happen O(n) times.

Running time

- In each call of M-ComputeOpt, there is a constant number of operations, besides the recursive calls. So the running time is bounded by the number of recursive calls.

- The two recursive call only happen when M[j] is empty.

- But when they happens, M[j] is no longer empty.

- So the recursively calls only happen O(n) times.

- The running time of M-ComputeOpt is O(n), assuming we are given the intervals as sorted by their finishing times, otherwise O($n \log n$), to sort them first.

So our algorithm …

- … solved the main problem by solving subproblems of smaller sizes,
- stored the solutions to the smaller problems in an array,
- recalled them from the array every time they needed to used. (memoization).
- Anything else?

What does M-ComputeOpt(n) actually find?

M-ComputeOpt(j)

If j=0 then
    Return 0

Else if M[j] is not empty then
    Return M[j]

Else
    M[j] = **max{**v(j) + **M-ComputeOpt**(p_j) **, M-ComputeOpt**(j- 1)**}**
    Return M[j]

EndIf

What does M-ComputeOpt(n) actually find?

M-ComputeOpt(j)

It finds the value of the optimal schedule O.

If j=0 then
    Return 0

Else if M[j] is not empty then
    Return M[j]

Else
    M[j] = **max{**v(j) + **M-ComputeOpt(**p_j**)** , **M-ComputeOpt(**j-1**)}**
    Return M[j]

EndIf

What does M-ComputeOpt(n) actually find?

M-ComputeOpt(j)

If j=0 then
  Return 0

Else if M[j] is not empty then
  Return M[j]

Else
  M[j] = **max**{v(j) + M-ComputeOpt(p_j) , M-ComputeOpt(j-1)}
  Return M[j]

EndIf

It finds the value of the optimal schedule O.

Is that what we were looking for?

Weighted Interval Scheduling

- A set of requests {*1*, *2*, … , *n*}.
  - Each request has a starting time $s(i)$, a finishing time $f(i)$, and a value $v(i)$.
  - Alternative view: Every request is an interval $[s(i), f(i)]$ associated with a value $v(i)$.
- Two requests *i* and *j* are compatible if their respective intervals do not overlap.
- Goal: Output a schedule which maximises the total value of compatible intervals.

Weighted Interval Scheduling

- A set of requests $\{1, 2, \ldots, n\}$.
  - Each request has a starting time $s(i)$, a finishing time $f(i)$, and a value $v(i)$.
  - Alternative view: Every request is an interval $[s(i), f(i)]$ associated with a value $v(i)$.
- Two requests *i* and *j* are compatible if their respective intervals do not overlap.
- Goal: Output a schedule which maximises the total value of compatible intervals.

Weighted Interval Scheduling

- A set of requests $\{1, 2, \ldots, n\}$.
  - Each request has a starting time $s(i)$, a finishing time $f(i)$, and a value $v(i)$.
  - Alternative view: Every request is an interval $[s(i), f(i)]$ associated with a value $v(i)$.
- Two requests i and j are compatible if their respective intervals do not overlap.
- Goal: Output a **schedule** which maximises the total value of compatible intervals.

From values to schedules

In other words, j is in O *if and only if*

$\mathrm{OPT}(p_j)+v(j) \geq \mathrm{OPT}(j-1)$

From values to schedules

In other words, j is in O *if and only if*

$OPT(p_j)+v(j) \geq OPT(j-1)$

FindSolution(j)

If j=0 , no solution

Else
  If $v(j) + M(p_j) \geq M(j-1)$ then
    Output j together with FindSolution(p_j)
  Else
    Output FindSolution($j-1$)
  EndIf
End If

From values to schedules

In other words, j is in O *if and only if*

$$OPT(p_j)+v(j) \geq OPT(j-1)$$

```
FindSolution(j)

  If j=0 , no solution

  Else
    If v(j) + M(pj) ≥ M(j-1) then
       Output j together with FindSolution(pj)
    Else
       Output FindSolution(j-1)
    EndIf
  End If
```

This can be done in $O(n)$ time.

Dynamic Programming vs Divide and Conquer

- DP is an optimisation technique and is only applicable to problems with optimal substructure.
- DP splits the problem into parts, finds solutions to the parts and joins them.
  - The parts are not significantly smaller and are overlapping.
- In DP, the subproblem dependency can be represented by a DAG.

- DQ is not normally used for optimisation problems.
- DQ splits the problem into parts, finds solutions to the parts and joins them.
  - The parts are significantly smaller and do not normally overlap.
- In DQ, the subproblem dependency can be represented by a tree.