Dynamic Programming

Part One

Announcements

- Problem Set Four due right now if you're using a late period.
    - Solutions will be released at end of lecture.
- Problem Set Five due Monday, August 5.
    - Feel free to email the staff list (**cs161-sum1213-staff@lists.stanford.edu**) with questions!
- Final project information will be announced early next week.
- A quick reminder about the Honor Code…

Outline for Today

- **Buying Cell Towers**
  - A surprisingly nuanced problem.
- **Dynamic Programming**
  - A completely different approach to recursion.
- **Weighted Activity Selection**
  - Breaking greedy algorithms, then fixing them.

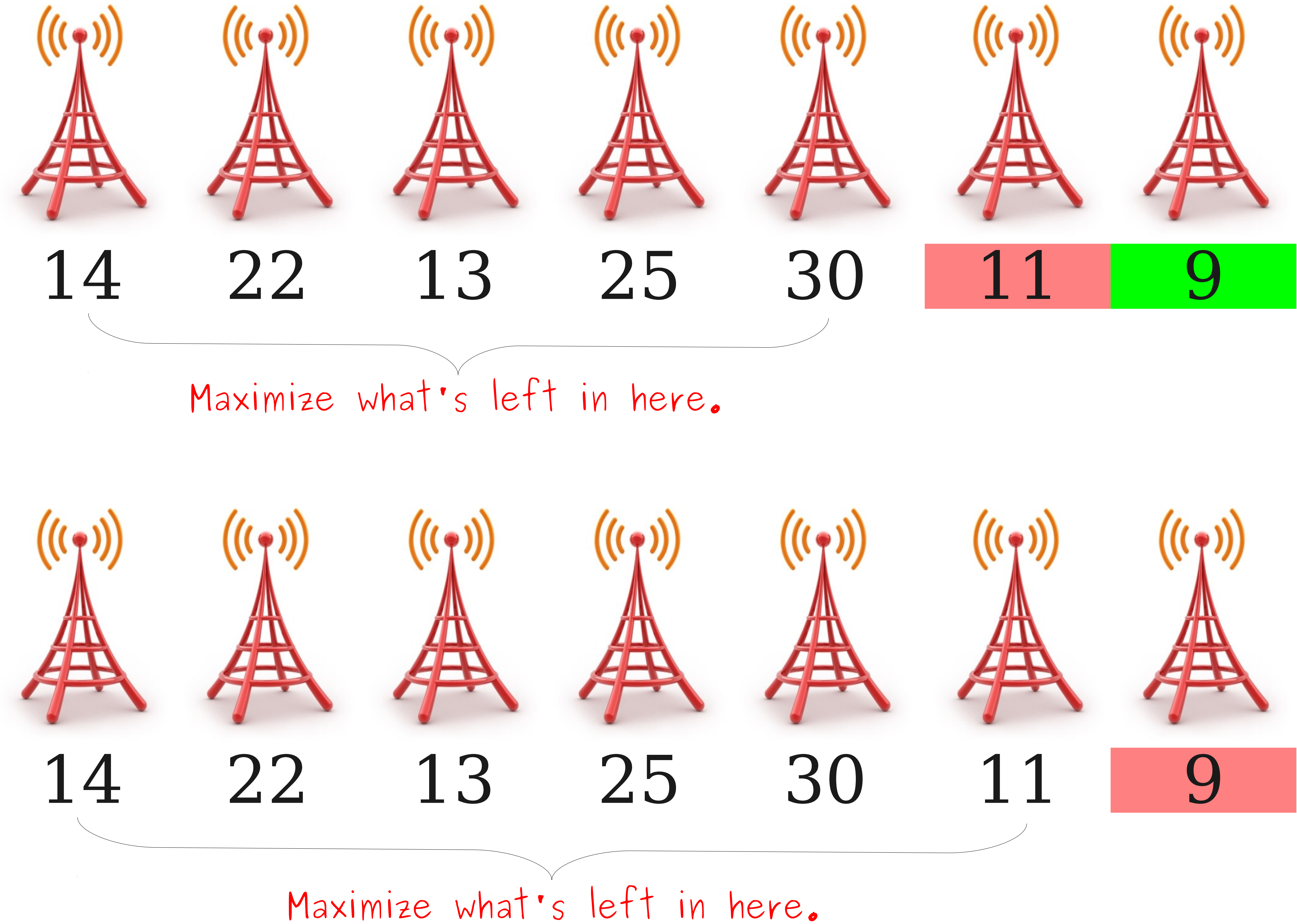
Example: Cell Tower Purchasing

Buying Cell Towers

137	42	95	272	52

The Cell Tower Problem

- You are given a list of town populations.
- You can build cell towers in any town as long as you don't build towers in adjacent cities.
- Two questions:
  - What is the largest number of people you can cover?
  - How do you cover them?


14 22 13 25 30 11 9
Maximize what's left in here.
14 22 13 25 30 11 9
Maximize what's left in here.

Some Notation

- Let v_k be the value of the kth cell tower, 1-indexed.
- Let OPT(k) be the maximum number of people we can cover using the first k cell towers.
- If C is a set of cell towers, let $C(k)$ denote the number of people covered by the towers in C numbered at most k.
- **Claim:** OPT(k) satisfies

$$\mathrm{OPT}(k) = \begin{cases} 0 & \text{if } k = 0 \\ v_k & \text{if } k = 1 \\ \max\{\mathrm{OPT}(k-1), v_k + OPT(k-2)\} & \text{otherwise} \end{cases}$$

Theorem: OPT(k) satisfies the previous recurrence.

Proof: If $k = 0$, no people can be covered, so OPT(0) = 0. If $k = 1$, we can choose tower 1 (value v_1) or no towers (value 0), so OPT(1) = v_1. So consider $k > 1$.

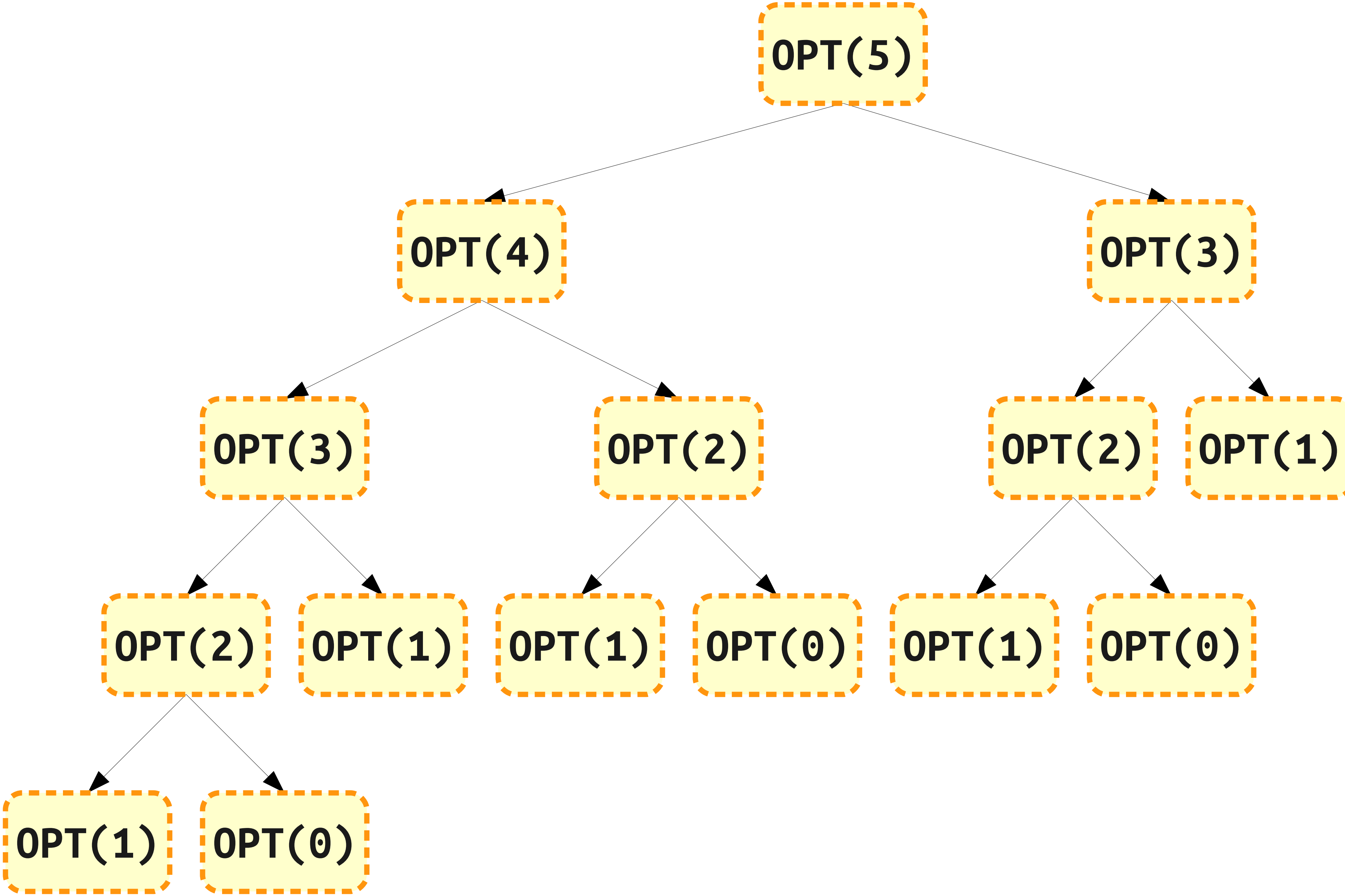
If $k \in C$, then $k - 1 \notin C$. Then all towers in C besides k are within the first $k - 2$ towers, so $C(k - 2) \leq$ OPT($k - 2$). Also, C(k – 2) ≥ OPT(k – 2); otherwise we could replace all towers in C except k with an optimal set of the first k – 2 towers to improve C. Thus OPT(k) = v_k + OPT(k – 2).

If $k \notin C$, all towers in C are in the first k – 1 towers. Thus $C(k - 1) \leq$ OPT(k – 1). Also, $C(k - 1) \geq$ OPT(k – 1); if not, we could improve C by replacing it with an optimal set of the first k – 1 towers. Therefore, OPT(k) = OPT(k – 1).

Since the optimal solution for k towers must be the better of these, OPT(k) = max{OPT(k – 1), v_k + OPT(k – 2)}. ■

A Simple Recursive Algorithm

- Here is a simple recursive algorithm for computing OPT(k):
  - If $k = 0$, return 0.
  - If $k = 1$, return v_k.
  - Return $\max\{\text{OPT}(k - 1), \text{OPT}(k - 2) + v_k\}$
- This follows directly from the recursive definition of OPT.
- **Question:** How efficient is this algorithm?


OPT(5)
OPT(4)
OPT(3)
OPT(3)
OPT(2)
OPT(2)
OPT(1)
OPT(2)
OPT(1)
OPT(1)
OPT(0)
OPT(1)
OPT(0)
OPT(1)
OPT(0)

A Problem

- The number of function calls made is given by this recurrence:

$$\boxed{\begin{aligned} &T(0) = 1 \\ &T(1) = 1 \\ &T(n) = T(n-1) + T(n-2) + 1 \end{aligned}}$$

- Can show that $T(n) = 2F_{n+1} - 1$, where F_{n+1} is the $(n + 1)$st Fibonacci number.
- $F_n = \Theta(\varphi^n)$, where $\varphi \approx 1.618...$ is the golden ratio.
- ***Runtime is exponential!***

Redundantly Redoing Completed Work That's Already Been Done

- This algorithm is inefficient because different branches of the recursion recompute the same work.
- Total number of *unique* recursive calls is low, though the total number of recursive calls is large.
- **Idea:** Avoid redundant work!
- How can we do this?

A Better Approach

- **Key Idea:** Compute answers *bottom-up* rather than *top-down*.
- Specifically:
  - Compute OPT(0) and OPT(1) directly.
  - Compute OPT(2) from OPT(0) and OPT(1).
  - Compute OPT(3) from OPT(1) and OPT(2).
  - Compute OPT(4) from OPT(2) and OPT(3).
  - …
  - Compute OPT(n) from OPT(n–1) and OPT(n–2)

Computing Bottom-Up

14 22 13 25 30 8 11

OPT(0)	OPT(1)	OPT(2)	OPT(3)	OPT(4)	OPT(5)	OPT(6)	OPT(7)
0	14	22	27	47	57	57	68

$$\mathrm{OPT}(k) = \begin{cases} 0 & \text{if } k=0 \\ v_k & \text{if } k=1 \\ \max\{\mathrm{OPT}(k-1),\, v_k + OPT(k-2)\} & \text{otherwise} \end{cases}$$

```
procedure maxCoverage(list A):
    let dp be a list of size length(A) + 1,
        zero-indexed.

    dp[0] = 0
    dp[1] = A[1]

    for i = 2 to length(A):
        dp[i] = max(dp[i - 1], A[i] + dp[i - 2])

    return dp[length(A)]
```

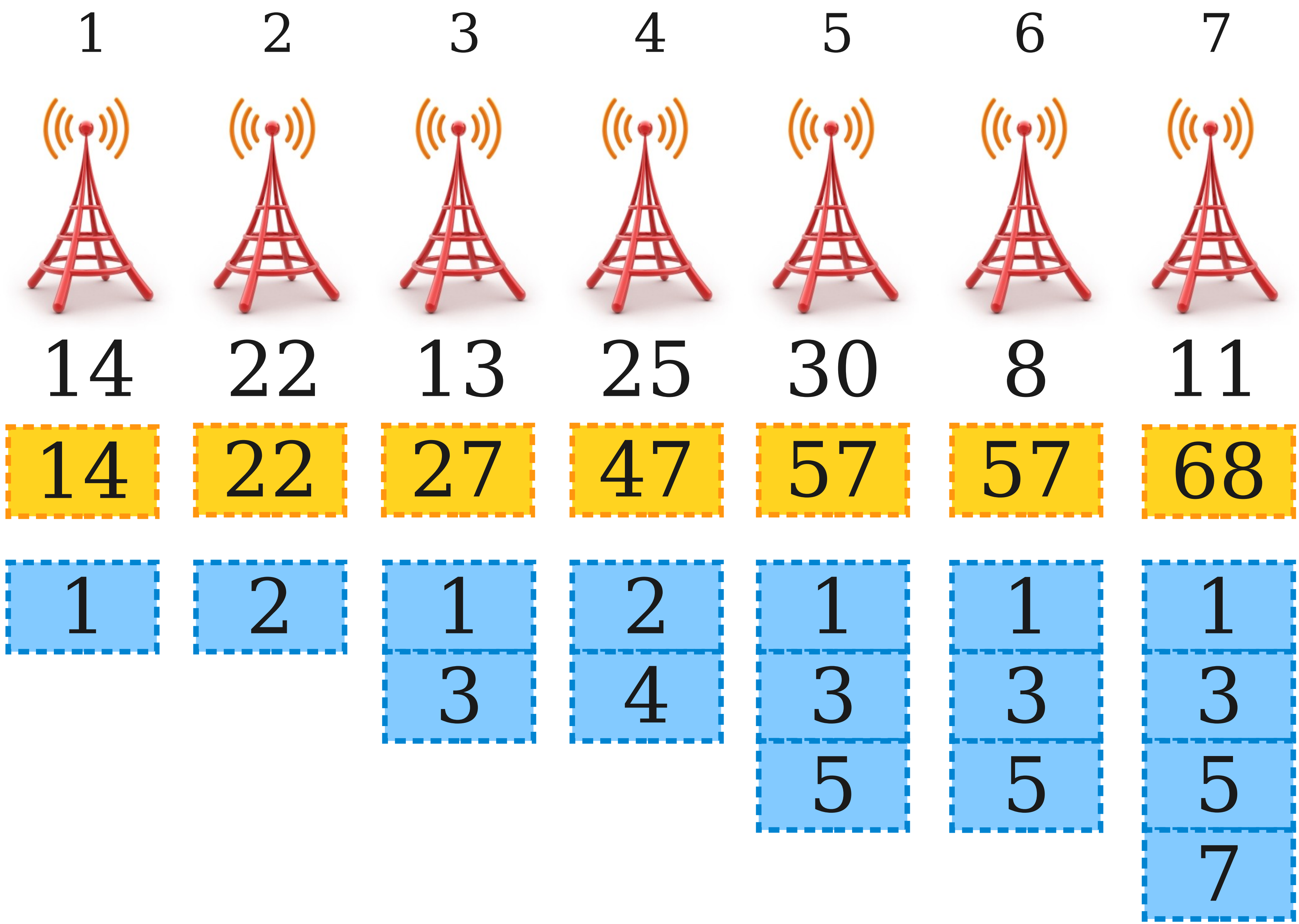
A Great Solution

- This new algorithm runs in time O(n) and works in O(n) space.
- Still evaluates the same subproblems, but does so only once and in a different order.
- This style of problem solving is called **dynamic programming**.

Dynamic Programming

- This algorithm works correctly because of the following three properties:
  - **Overlapping subproblems:** Different branches of the recursion will reuse each other's work.
  - **Optimal substructure:** The optimal solution for one problem instance is formed from optimal solutions for smaller problems.
  - **Polynomial subproblems:** The number of subproblems is small enough to be evaluated in polynomial time.
- A **dynamic programming** algorithm is one that evaluates all subproblems in a particular order to ensure that all subproblems are evaluated only once.

Recovering the Solution



1
2
3
4
5
6
7
14
22
13
25
30
8
11
14
22
27
47
57
57
68
1
2
1
3
2
4
1
3
5
1
3
5
1
3
5
7

An Initial Approach

- Our original algorithm uses $O(n)$ time and $O(n)$ space.
- This new approach might use $\Theta(n^2)$ space just storing the incremental optimal solutions.
- It also might take $\Theta(n^2)$ time copying answers down the line.
- Can we do better?

14	22	13	25	30	8	11
14	22	27	47	57	57	68

Recovering the Solution

- Once you have filled in a DP table with values from the subproblems, you can often reconstruct the optimal solution by running the recurrence backwards.
- This is often done with a greedy algorithm, since the algorithm will never get stuck anywhere.
  - Consequence of the fact that you know the true values of all subproblems.

Reducing Space Usage

- If you only need the *value* of the optimal answer, can save space by not storing the whole table.
- For cell towers, all DP values depend only on previous two elements.

```
procedure maxCellTowers(list A):
    let a = 0
    let b = A[1]

    for i = 2 to length(A):
        let newVal = max(a + A[i], b)
        a = b
        b = newVal

    return b
```

A Second Example:
Weighted Activity Selection

Weighted Activity Scheduling

- Not all fun activities are equally fun!
- Given a set of activities, *which have associated weights*, choose the set of non-overlapping activities that will maximize the total weight.
- A more realistic generalization of the problem we saw earlier.

An Algorithmic Insight

- Sort the activities in ascending order of finish time, breaking ties arbitrarily.
- The optimal solution either
  - Includes the very last event to finish, in which case it chooses an optimal set of activities from the activities that don't overlap it.
  - Doesn't include it, in which case it can choose from all other activities.

Formalizing the Idea

- Number the activities $a_1, a_2, \ldots, a_n$ in ascending order of finishing time, breaking ties arbitrarily. Let w_k denote the weight of a_k.
- Let $p(i)$ represent the *predecessor* of activity a_i (the latest activity a_k where a_k ends before a_i starts). If there is no such activity, set $p(i) = 0$.
- Let OPT(k) be the maximum weight of activities you can schedule using the first k activities.
- For any schedule S, let $S(k)$ denote the weight of all activities in S numbered at most k.
- **Claim:** OPT(k) satisfies the recurrence

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

Theorem: OPT(k) satisfies the previous recurrence.

Proof: If k = 0, OPT(0) = 0 since there are no activities. So consider k > 0.

If $a_k \notin S$, then S consists purely of activities drawn from the first k – 1 activities. Thus $S(k - 1) \leq$ OPT(k – 1). Moreover, $S(k - 1) \geq$ OPT(k – 1), since otherwise we could replace S with an optimal solution for the first k – 1 activities to improve upon it. Thus $S(k)$ = OPT(k – 1).

If $a_k \in S$, then no activity a_m where $p(k) < m < k$ can be in S, since these activities overlap a_k. Since all activities in S other than a_k are chosen from the first $p(k)$ activities, $S(p(k)) \leq$ OPT($p(k)$). Also, $S(p(k)) \geq$ OPT($p(k)$) (if not, we could improve S by replacing these activities with an optimal solution for the first $p(k)$ activities.) Therefore, $S(k) = w_k$ + OPT($p(k)$).

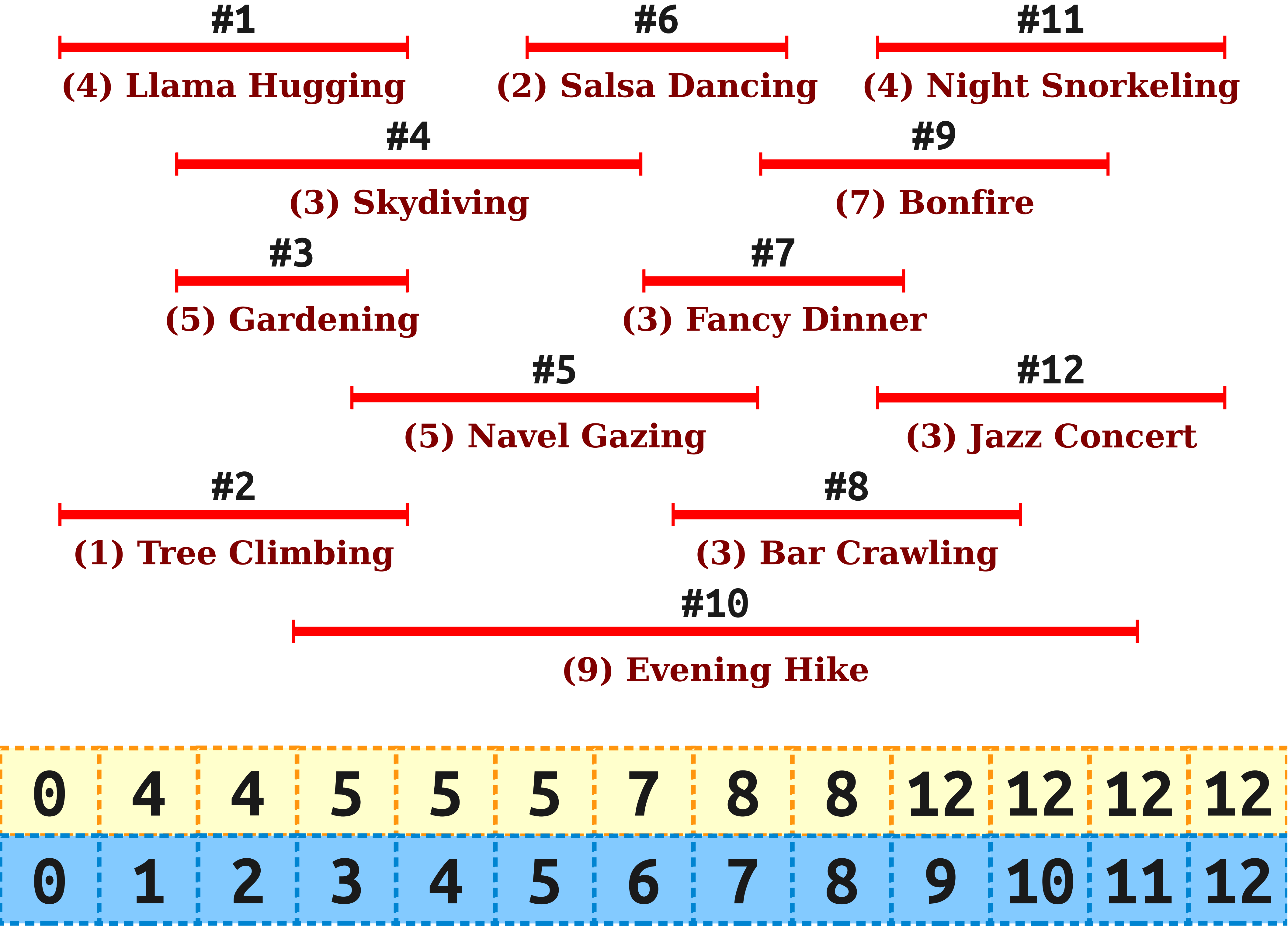
Since OPT(k) must be the better of these two options, we have that OPT(k) = max{OPT(k – 1), w_k + OPT($p(k)$)} ■

Cut-and-Paste Arguments

- The style of argument used in the previous proof is sometimes called a ***cut-and-paste argument***.
  - To show optimal substructure, assume that some piece of the optimal solution S^* is not an optimal solution to a smaller subproblem.
  - Show that replacing that piece with the optimal solution to the smaller subproblem improves the allegedly optimal solution S^*.
  - Conclude, therefore, that S^* must include an optimal solution to a smaller subproblem.
- This style of argument will come up repeatedly when discussing dynamic programming.

Evaluating the Recurrence

- As before, evaluating this recurrence directly would be enormously inefficient.
- Why?
- **Overlapping subproblems!**
  - Multiple different branches of the computation all will make the same calls.
- Instead, as before, we can evaluate everything bottom-up.

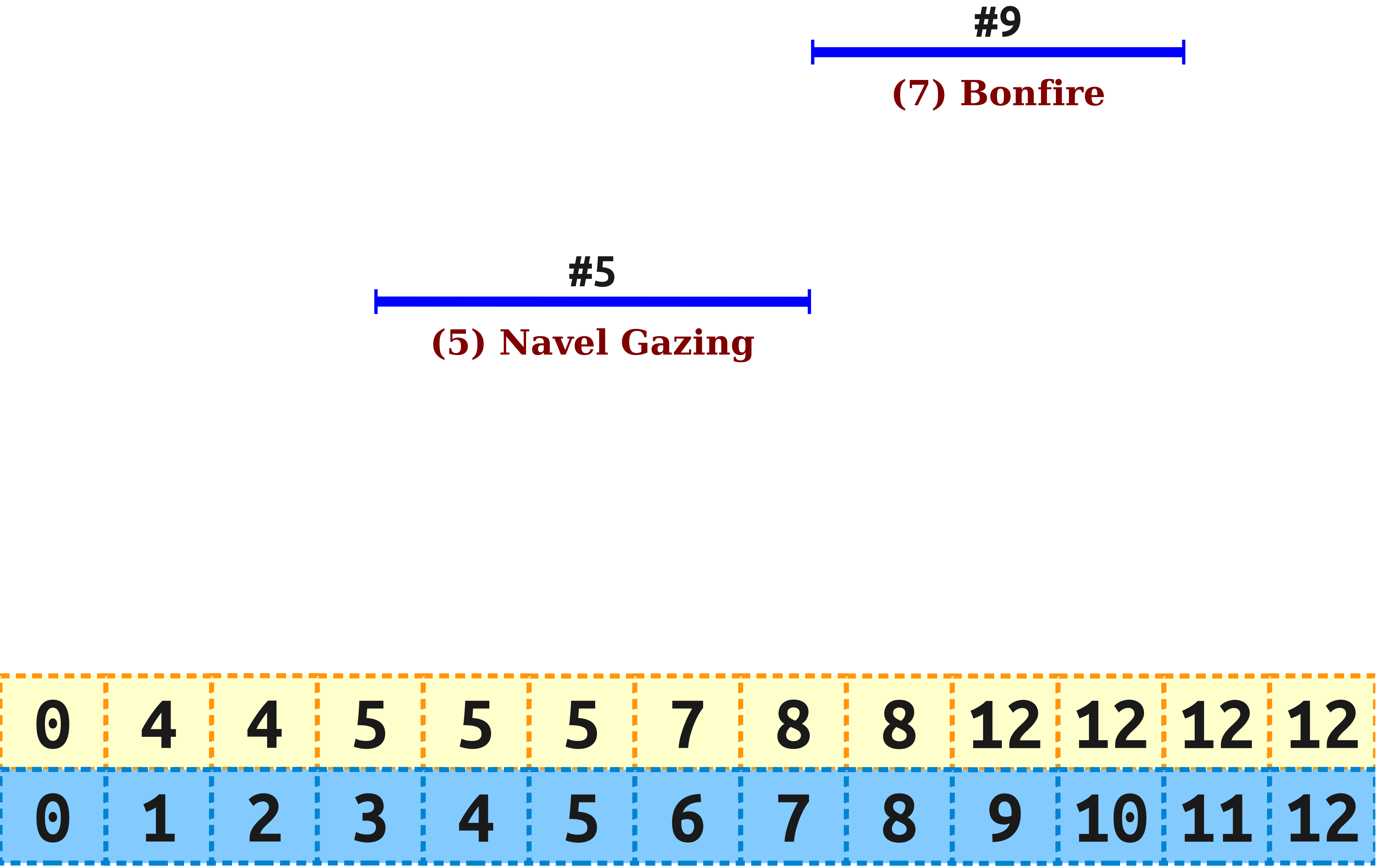

#1
(4) Llama Hugging
#6
(2) Salsa Dancing
#11
(4) Night Snorkeling
#4
(3) Skydiving
#9
(7) Bonfire
#3
(5) Gardening
#7
(3) Fancy Dinner
#5
(5) Navel Gazing
#12
(3) Jazz Concert
#2
(1) Tree Climbing
#8
(3) Bar Crawling
#10
(9) Evening Hike
0 4 4 5 5 5 7 8 8 12 12 12 12
0 1 2 3 4 5 6 7 8 9 10 11 12

```
procedure weightedActivitySelection(list A):
    let dp be an array of size length(A) + 1,
        0-indexed.

    dp[0] = 0

    for i = 1 to length(A):
        dp[i] = max(A[i] + dp[p(i)], dp[i - 1])

    return dp[length(A)]
```


#9
(7) Bonfire
#5
(5) Navel Gazing
0 4 4 5 5 5 7 8 8 12 12 12 12
0 1 2 3 4 5 6 7 8 9 10 11 12

Why This Works

- As before, this problem exhibits three properties:
  - **Overlapping subproblems**: Many different recursive branches have the same subproblems.
  - **Optimal substructure**: The solution for size n depends on the optimal solutions for smaller sizes.
  - **Polynomial subproblems**: There are only $O(n)$ total subproblems.
- This is why the DP solution works.

Next Time

- Sequence Alignment
- The Needleman-Wunsch Algorithm
- Levenshtein Distance