“Those who cannot remember the past are condemned to repeat it.”

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás

Dynamic programming
**Definition.** Fibonacci numbers are defined by the following recurrence:

\[
F_n = \begin{cases} 
0 & \text{if } n = 0 \\ 
1 & \text{if } n = 1 \\ 
F_{n-1} + F_{n-2} & \text{otherwise}
\end{cases}
\]

Recall three different implementations of Fibonacci from our activity on Wednesday:

- Naively recursive
- Local array to “memoize” the first \( n \) numbers
- Global array, worked backwards from \( n \)
The naive recurrence was horribly sloooooow

**RecFIBO**\((n)\):

- if \(n = 0\) return 0
- else if \(n = 1\) return 1
- else return **RecFIBO**\((n - 1) + **RecFIBO**\((n - 2)\)

- **Practice:** can we lower bound the cost?
- Step 1: Write the recurrence

\[
T(n) = T(n - 1) + T(n - 2) + O(1)
\]
Can we lower bound the running time using techniques we already know?

\[ T(n) = T(n - 1) + T(n - 2) + \Theta(1) \]

- If we want to show that \( a \geq c \), we can show \( a \geq b \) and \( b \geq c \)

\[ T(n) \geq 2T(n - 2) + \Omega(1) \]

Let's draw this tree!

- There are \( n/2 \) levels, each level has \( 2^i \) nodes
- Level \( i \) has cost \( \Omega(2^i) \)

\[ T(n) = \Omega(2^{n/2}) \]
• Recursive Fibonacci algorithm is slow because it recomputes the same functions over and over.

• We saw that we can speed it up considerably by writing down the results of our recursive calls, and looking them up when we need them later.
Dynamic Programming: Smart Recursion

- Dynamic programming is all about smart recursion by using memoization.
- Here (fib3 from activity) we cut down on all useless recursive calls.


Green arrows store the results.

Orange arrows read stored results.
Dynamic Programming: Recursion + Memoization

- **Memoization**: technique of storing expensive function call results so that they can be looked up later

- To be useful, we must carefully structure our algorithm to traverse problem space in appropriate order

- Memoization is a core concept of dynamic programming, but also used elsewhere
Recipe for a Dynamic Program

- **Formulate the right subproblem.** The subproblem must have an optimal substructure.

- **Formulate the recurrence.** Identify how the results of the smaller subproblems can contribute to results of larger subproblems.

- **State the base case(s).** The subproblem(s) so small we know the answer immediately!

- **State the final answer.** (In terms of the subproblem(s))

- **Choose a memoization data structure.** Where are you going to store already computed results? (This is often a “table”)

- **Identify evaluation order.** Identify the dependencies: which subproblems depend on which? Using these dependencies, identify an evaluation order.

- **Analyze space and running time.** As always!
Weighted Scheduling

Further Reading: Chapter 6, KT
**Weighted Scheduling**

**Job scheduling.** Suppose you have a machine that can run one job at a time; \( n \) job requests, where each job \( i \) has a start time \( s_i \), finish time \( f_i \) and weight \( v_i \geq 0 \).

Overlapping jobs e.g., d and g are incompatible
Weighted Scheduling

Input. Given \( n \) intervals labeled \( 1, \ldots, n \) with starting and finishing times \( \{(s_1, f_1), \ldots, (s_n, f_n)\} \) and non-negative weights \( \{v_1, \ldots, v_n\} \).

Goal. We must select compatible (non-overlapping) intervals with the maximum weight.

• That is, our goal is to find a set of intervals \( I \subseteq \{1, \ldots, n\} \) that are pairwise non-overlapping and that maximize \( \sum_{i \in I} v_i \).
Remember Greedy?

- In Unweighted, *earliest-finish-time first* was optimal greedy algorithm
  - Consider jobs in order of finish times
  - Greedily pick jobs that are non-overlapping
- We proved greedy is optimal when all weights are 1
- How about the weighted interval scheduling problem?

**Greedy fails spectacularly!**
Different Greedy?

We saw that not it is important to choose the right thing to “be greedy” over. Should we just pick other optimization criteria?

• **New idea:** greedily select intervals with the maximum weights, remove overlapping intervals

• Does this work?

![Graph showing Greedy fails spectacularly!](image-url)
Let’s Think Recursively

The heart of dynamic programming is recursively thinking.

• Coming up with a **smaller subproblem** that has the **same optimal structure** as the original problem.

• First, let’s focus on the total **value** of the optimal solution, rather than the actual set of intervals. That is,

• **Optimal value:**
  The largest $\sum_{i \in I} v_i$ where intervals in $I$ are compatible.

• Let’s also define $\text{Opt-Schedule}(n)$ to be the **value** of the optimal schedule that considers the first $n$ intervals.
Let’s Think Recursively

Consider the last interval: it’s either in the optimal solution or it’s not.

- Whatever the optimal solution is, we can find it by considering both cases (in or out) and taking their maximum weight.

- **Case 1.** Last interval **is not** in the optimal solution
  - Remove the last interval.
    We now have a smaller subproblem!

- **Case 2.** Last interval **is** in the optimal solution
  - Anything that overlaps with this interval cannot be in the solution. Remove them.
    We now have a smaller subproblem!
Formalize the Subproblem

**Opt-Schedule**$(i)$: value of the optimal schedule that only considers intervals $\{1,\ldots, i\}$, for $0 \leq i \leq n$
Base Case & Final Answer

**Opt-Schedule**\((i)\): value of the optimal schedule that only considers intervals \(\{1, \ldots, i\}\), for \(0 \leq i \leq n\)

**Base Case.** Opt-Schedule\((0)\) = 0

**Goal** (Final answer.) Opt-Schedule\((n)\)
Recurrence

How do we go from one subproblem to the next?

- The recurrence describes how to compute \( \text{Opt-Schedule}(i) \)
  by using values of \( \text{Opt-Schedule}(j) \) where \( j < i \)

**Case 1.** Say interval \( i \) is not in the optimal solution, can we write the recurrence for this case?

- \( \text{Opt-Schedule}(i) = \text{Opt-Schedule}(i - 1) \)
Recurrence

How do we go from one subproblem to the next?

• The recurrence describes how to compute $\text{Opt-Schedule}(i)$ by using values of $\text{Opt-Schedule}(j)$ where $j < i$

**Case 2.** Say interval $i$ is in the optimal solution, what is the smaller subproblem we should recurse on for this case?

• No interval $j < i$ that overlaps with $i$ can be in solution

• Need to remove all such intervals to get our smaller subproblem

• How do we do that?
Suppose the intervals are sorted by finish times.

- Let $p(j)$ be the predecessor of $j$. That is, largest index $i < j$ such that intervals $i$ and $j$ are not overlapping.
- Define $p(j) = 0$ if all intervals $i < j$ overlap with $j$.
Helpful Information

Let $p(j)$ be the predecessor of $j$. That is, largest index $i < j$ such that intervals $i$ and $j$ are not overlapping.

• $p(8) = ?$, $p(7) = ?$, $p(2) = ?$
Let $p(j)$ be the predecessor of $j$. That is, largest index $i < j$ such that intervals $i$ and $j$ are not overlapping.

- $p(8) = 1$, $p(7) = 3$, $p(2) = 0$
Recurrence

How do we go from one subproblem to the next?

- The recurrence describes how to compute \( \text{Opt-Schedule}(i) \) by using values of \( \text{Opt-Schedule}(j) \) where \( j < i \)

**Case 2.** Say interval \( i \) is in the optimal solution, what is the smaller subproblem we should recurse on for this case?

  - Suppose we know \( p(i) \) (the predecessor of \( i \)), how can we write the recurrence for this case?
  - \( \text{Opt-Schedule}(i) = \text{Opt-Schedule}(p(i)) + v_i \)
DP Recurrence

\[
\text{Opt-Schedule}(i) = \max\{\text{Opt-Schedule}(i - 1), v_i + \text{Opt-Schedule}(p(i))\}
\]

Optimal schedule that excludes interval \(i\)

Optimal schedule that includes interval \(i\)
Filling Out the DP Table
Filling Out the DP Table

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

time

7

10

2

2

3

8

4
### Filling Out the DP Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Diagram

- **Time**
  - 0
  - 1
  - 2
  - 3
  - 4

- **Events**
  - 7
  - 10
  - 8
  - 2

- **Activities**
  - 10
  - 2
  - 4
  - 3
# Filling Out the DP Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

![Diagram showing the filling of the DP table with time intervals and corresponding actions or values]
Summary of DP

- **Subproblem.** Formulate the optimal substructure
  - For $0 \leq i \leq n$, let $\text{Opt-Schedule}(i)$ be the value of the optimal schedule that only uses intervals $\{1, \ldots, i\}$

- **Recurrence.** How to go from one subproblem to the next
  - $\text{Opt-Schedule}(i) = \max \{ \text{Opt-Schedule}(i - 1), v_i + \text{Opt-Schedule}(p(i)) \}$

- **Base case.** The problem(s) we immediately know the answer to.
  - $\text{Opt-Scheduler}(0) = 0$ (no intervals to schedule)

- **Correctness.**
  - Use induction based on the recurrence
Remaining Pieces

• Final answer in terms of subproblem?
  • Opt-Schedule[$n$]

• Evaluation order (in what order can be fill the DP table)
  • $i = 0 \rightarrow n$, start with base case and use that to fill the rest

• Memoization data structure: 1-D array

• Final piece:
  • Running time and space
  • Space: $O(n)$
  • Time: preprocessing + time to fill array
Computing $p[i]$ (Preprocessing)

- How quickly can we compute $p[i]$?
  - We could do a linear scan for each $i$: $O(i)$ per interval
  - This would be $O(n^2)$ overall...

- What if we had intervals sorted by their finish time $F[1,\ldots,n]$?
  - For each interval, we could binary search over $F[1,\ldots,n]$ to find the first $j < i$ such that $f_j \leq s_i$
    - Binary searching would take $O(\log n)$ per interval, $O(n \log n)$ total
  - Time $O(n \log n)$ to compute the array $p[]$
    - This covers sorting + binary searching
Running Time

• How many subproblems do we need to solve?
  • $O(n)$

• How long does it take to solve a subproblem?
  • $O(1)$ to take the max

• Preprocessing time:
  • Need to sort; $O(n \log n)$
  • Need to find $p(i)$ for all each $i$: $O(n \log n)$

• Overall: $O(n \log n) + O(n) = O(n \log n)$

Wait!!! We’ve only computed the value, not the actual interval set!!!
Recreating Chosen Intervals

Suppose we have \( M[] \) of optimal weights.

- **Big Question**: How can we reconstruct the optimal set of intervals?

Identifying which of the two cases was larger tells us whether or not interval \( i \) was included:

\[
\text{Opt-Schedule}(i) = \max\{\text{Opt-Schedule}(i - 1), v_i + \text{Opt-Schedule}(p(i))\}
\]

This value is bigger: \( i \) not in OPT

This value is bigger: \( i \) is in OPT
Recursive Solution?

Suppose for now that we do not memoize: just a divide and conquer recursion approach to the problem.

Opt-Schedule($i$):

- If $j = 0$, return 0
- Else
  - Return $\max(\text{Opt-Schedule}(j - 1), v_j + \text{Opt-Schedule}(p(j)))$

- How many recursive calls in the worst case?
  - Depends on $p(i)$
- Can we create a really bad instance?
Recursive Solution: Exponential

- For this example, asymptotically how many recursive calls?
- Grows like the Fibonacci sequence (exponential):
  \[ T(n) = T(n - 1) + T(n - 2) + O(1) \]
- Lots of redundancy!
  - How many distinct subproblems are there to solve?
  - \text{Opt-Schedule}(i) \text{ for } 1 \leq i \leq n + 1

\[ p(1) = 0, \ p(j) = j-2 \]

\[ \text{recursion tree} \]
Dynamic Programming Tips

• Recurrence/subproblem is the key!

• DP is a lot like divide and conquer, while writing extra things down

• When coming to a new problem, ask yourself what subproblems may be useful? How can you break that subproblem into smaller subproblems?

• Be clear while writing the subproblem and recurrence!

• In DP we usually keep track of the cost of a solution, rather than the solution itself
Acknowledgments

• Some of the material in these slides are taken from
  
  
  • Jeff Erickson’s Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)