1 Dynamic Programming

The Divide & Conquer paradigm worked by dividing the problem into, typically, non-overlapping subproblems of the same type. The idea can also be used when subproblems overlap extensively, as long as care is taken to avoid unneeded duplication of computation.

A simple example of a problem that benefits from avoiding unneeded recursion is computing the \( n^{th} \) Fibonacci number.

\[
\text{fib}(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
\text{fib}(n - 1) + \text{fib}(n - 2) & \text{otherwise}
\end{cases}
\]

Note that

1. \( \text{fib} \) is defined recursively, so it naturally divides work into subproblems.
2. \( \text{fib} \) involves two recursive calls, each of which calls \( \text{fib}(n - 3) \) at some point, so it exhibits overlapping subproblems.
3. In fact, the number of recursive calls is exponential!
   - Let \( C(n) \) be the number of calls that occur when \( \text{fib}(n) \) is invoked
   - Then
     \[
     C(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
1 + C(n - 1) + C(n - 2) & \text{otherwise}
\end{cases}
\]
   - So, \( C(n) \geq \text{fib}(n) \) for all \( n \geq 1 \).

Memoizing We can (dramatically) improve the run time of \( \text{fib} \) by avoiding the repeated computation of \( \text{fib}(i) \).

Suppose we create a table \( F[1..n] \) that stores fibonacci numbers the first time they are computed (and 0 until then).

Then we can write \( \text{fib} \) as

**Algorithm 1** Fibonacci with Memoizing

```
procedure MEMOFib(n)// Prior to first call, \( F[1..n] \) has been set to 0
  if \( F[n] > 0 \) then
    return \( F[n] \)
  else if \( n = 1, 2 \) then
    \( F[n] = 1 \)
    return \( F[n] \)
  else
    \( F[n] = \text{memoFib}(n - 1) + \text{memoFib}(n - 2) \)
    return \( F[n] \)
end procedure
```

When \( \text{memoFib}(n) \) is invoked a sequence of \( n \) recursive calls will be made, after which the table \( F[] \) will begin to be populated. After this, all future calls will directly use \( F[] \). DRAW A TREE OF CALLS; FOLLOW RECURSION.

Often, as in this case, it makes sense to simply build the table directly, instead of with the recursive model.
Algorithm 2 Fibonacci Table

\begin{algorithm}
\begin{procedure}{\textsc{FIBTable}(n)}
\For{i ← 3 \text{ to } i ← n}{
F[i] = F[i - 1] + F[i - 2]
}\end{procedure}
\end{algorithm}

The Principle of Optimality

Many optimization problems exhibit what is often called the \textit{Principle of Optimality}, namely, that portions of an optimal solution to a problem must themselves be optimal solutions to the subproblem they generate. Examples include:

- Shortest paths: Any subpath of a shortest path is a shortest path.
- Spanning trees: Any subtree of a minimum cost spanning tree must be a minimum-cost tree.

Situations in which the principle of optimality holds are good candidates for a solution based on building and storing solutions to optimal subproblems—as long as not too many subproblems need to be solved!

Let’s do some examples

Weighted Interval Scheduling

The Problem:

- A single resource processes jobs, each of which has a start and end time \((s_i, t_i)\).
- The resource can handle one job at a time.
- Each resource has a positive value \(v_i\).
- The goal is to find a non-overlapping set of jobs of maximum value (sum of individual job value).

Can’t We Be Greedy?

Apparently not, no one has discovered a natural, greedy approach.

Does the Principle of Optimality Hold?

First, let’s assume that we have sorted the jobs by increasing finish time \(t_i\).

Suppose we define \(\text{maxSched}(n)\) to be the value of the optimal solution. If this solution doesn’t use job \(n\), then \(\text{maxSched}(n) = \{\text{maxSched}(n - 1)\}\). OPTIMALITY PRINCIPLE!

What if the solution \textit{does} use job \(n\)?

Let \(p(n)\) be the largest \(j < n\) such that \(t_j < s_n\) (\(j\) stops before \(n\) starts). Note that any solution that includes job \(n\) cannot contain any job \(j > p(n)\) (they overlap job \(n\)).

So, if \(\text{maxSched}(n)\) uses job \(n\), then \(\text{maxSched}(n) = v_n + \text{maxSched}(p(n))\). AGAIN, OPTIMALITY PRINCIPLE!

This gives

\[
\text{maxSched}(n) = \max\{\text{maxSched}(n - 1), v_n + \text{maxSched}(p(n))\}
\]

So, to design our algorithm, we need to build the \text{maxSched} table and compute \(p()\)-values.
To compute \( p() \)-values, imagine that we have sorted the combined start and finish times in increasing order. We will keep track of the index of the largest end time \( t = j \) seen so far.

- Let \( p[1] = 0, t = 0 \).
- While there are unscanned items in the list
  - Consider the next item in the list, call it \( x \).
  - If \( x \) is a start time, then \( x = s_k \) for some \( k \); set \( p[k] = t \).
  - If \( x \) is an end time, then \( x = t_k \) for some \( k \). Update \( t \) to be \( k \).

Thus \( p[] \) can be built in linear time, after sorting all of the start and end times, and, from this list, we can cull the order of the intervals sorted by end time, as needed by the algorithm.

Now \( maxSched \) can be built either recursively using memoizing or it can be constructed iteratively.

**Algorithm 3** MaxSched with Memoizing

```plaintext
procedure MAXSCHED(n)
  // Prior to first call, \( p[1..n] \) has been constructed
  // And a table \( M[1..n] \) has been initialized to 0
  if \( n = 0 \) then
    return 0
  else if \( M[n] > 0 \) then
    return \( M[n] \)
  else
    \( M[n] = \max\{maxSched(n - 1), v_n + maxSched(p[n])\} \)
    return \( M[n] \)
end procedure
```

**Algorithm 4** Iterated MaxSched

```plaintext
procedure MAXSCHED(n)
  // Prior to first call, \( p[1..n] \) has been constructed
  \( M[0] = 0 \)
  for \( i \leftarrow 1 \) to \( i \leftarrow n \) do
    \( M[i] = \max\{M[i - 1], v_i + M[p[i]]\} \)
  end procedure
```

As can be seen from the code, the table can be built in linear time once the function \( p() \) is known; \( p() \) can be constructed in \( O(n) \) time after sorting all start/end points in \( O(n \log n) \) time.

**But What If We Want the Actual Set of Intervals?**

An optimal set of intervals can be efficiently computed in one of two ways.

- Unwind the table \( M[] \): Note that if \( M[n - 1] < v_n + M[p[n]] \), then some optimal solution contains job \( n \); recursive we can then check \( M[p[n]] \) to see if job \( p[n] \) was used, etc. Otherwise, some optimal solution does not use job \( n \) and so we can check \( M[n - 1] \) to see if job \( n - 1 \) was used, etc.

- Build the solution as we go: Allow \( M[i] \) to store not only the value of the optimal solution for jobs 1, \ldots, \( i \), but also a flag indicating whether job \( i \) was part of that solution.
Why ”Dynamic Programming”?


"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. 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, lets kill two birds with one stone. Lets 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 is its 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.”