CS2521: Dynamic Programming

N. Oren
n.oren@abdn.ac.uk

University of Aberdeen

Fibonacci Sequence

- Recall the Fibonacci Sequence, where $F(0) = 0, F(1) = 1$

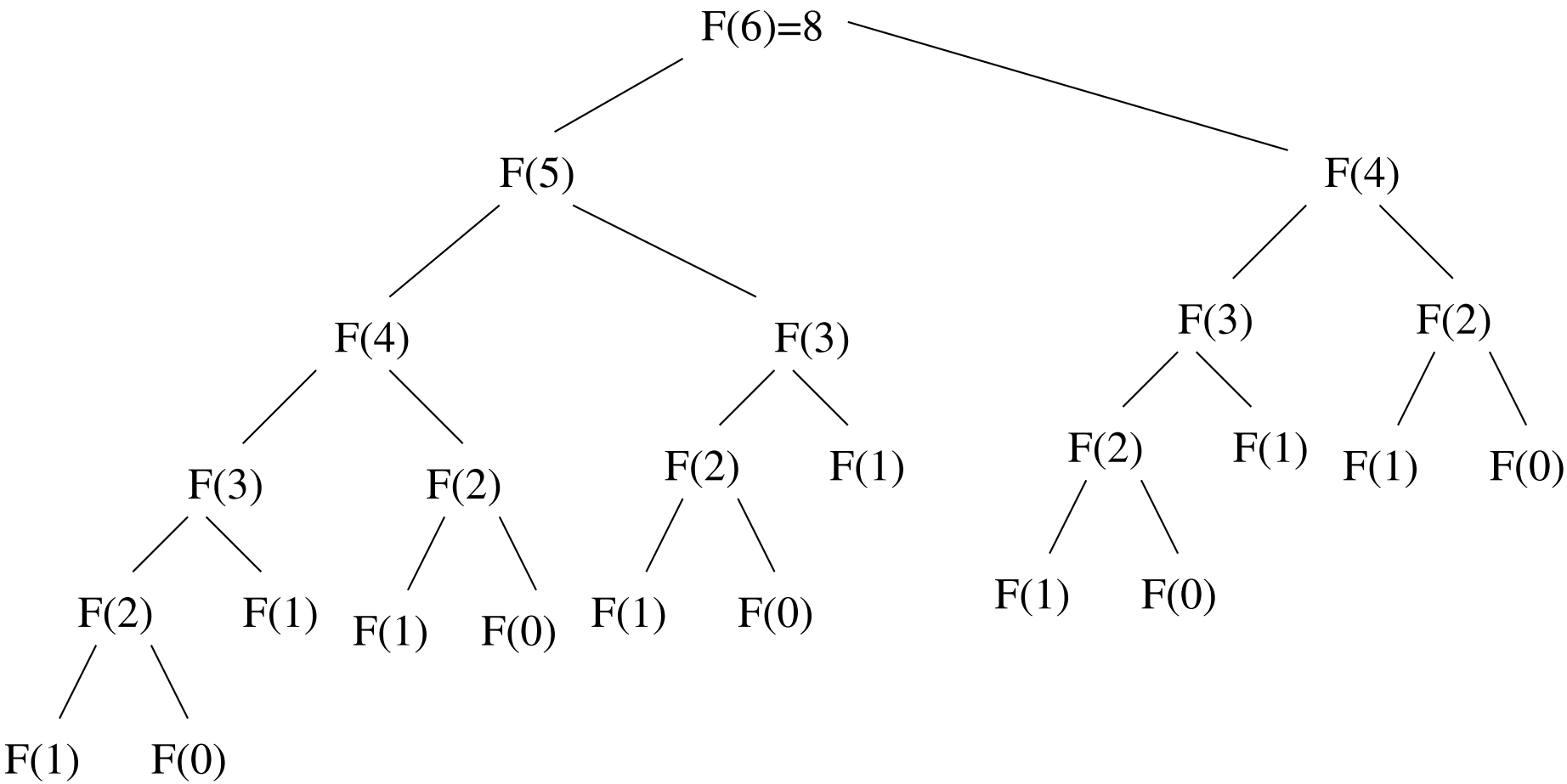
$$F(n) = F(n-1) + F(n-2)$$

- We can easily calculate this recursively.

```
function Fib(()n)
    if n=0 then
        return 0
    if n=1 then
        return 1
    return Fib(n-1)+Fib(n-2)
```

- Computing 45 Fibonacci numbers takes minutes. Why?

Fibonacci Sequence



Observations

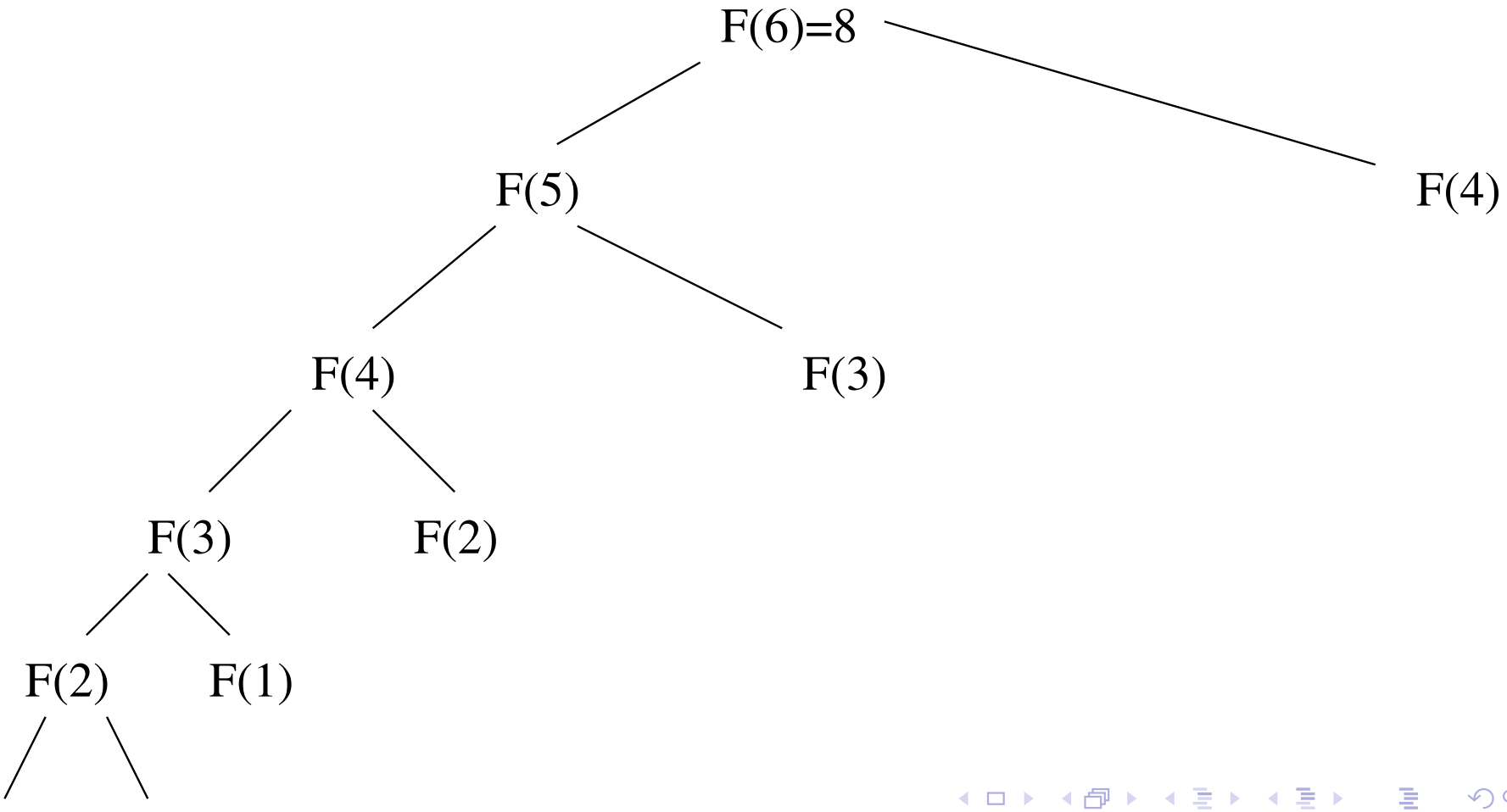
- $F(4)$ occurs twice
- $F(2)$ occurs five times
- In general, $F(n+1)/F(n)$ is approximately 1.6. Since our leaves contain only 0s and 1s, we must have at least 1.6^n leaves to compute $F(n)$
- So our program takes exponential time to run.

An improvement

C= an array of size n containing -1
$C[0] = 0$
$C[1] = 1$
function Fib2(n)
    if $C[n] = -1$ **then**
        C[n]= Fib2($n-1$)+Fib2($n-2$)
    return C[n]

What's happening?

- We are now caching values we have computed, allowing us to reuse them later on.
- Running time is $O(n)$



Caching

- In general, caching is a very effective strategy, and gives most of the benefits of dynamic programming (whatever that is).
- This strategy trades off computation time with storage space (in this case, linear space saves us exponential time, an excellent trade-off).
- We can do even better by explicitly specifying the order of evaluation of the recurrence relation.

Dynamic Fibonacci

function DynamicFib(n)
    i=0
    C=an array of size $|n|$
    C[0]=0
    C[1]=1
    for $i = 2$ to n **do**
        $C[i] = C[i-1] + C[i-2]$
    return $C[n]$

Observations

- We've got rid of the recursion, and the approach is still linear.
- But we can do even better, as we know precisely which values we need to calculate the sequence

Dynamic Fibonacci

```
function DynamicFib(n)
    back2=0
    back1=1
    currentSum;
    if n=0 then
        return 0
    for i = 2 to n − 1 do
        currentSum=back1+back2
        back2=back1
        back1=currentSum
    return back1+back2
```

Observations

- We are now using constant space, meaning no impact of array resizing etc.
- This is the heart of dynamic programming - smart caching of recursive functions to help running time.
- The term dynamic programming has nothing to do with programming as we know it
  - The "programming" is storing/retrieving values from a table.
  - The "dynamic" because we are building the table as we go along.

Binomial Coefficients

- Binomial coefficients appear in many combinatorial problems, counting the number of ways to choose k things out of n possibilities.

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

- Problem 1: n! (for example) quickly becomes very large, and will overflow many number representations, even if the binomial fits the representation.
- Alternative is a recurrence relation

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- Base cases:
  - For the left, we end up with $\binom{n-k}{0}$ — choose 0 things from a set, so equals 1.
  - For right, we end up with $\binom{k}{k}$ — again, 1.

Binomial Coefficients

1: **function** Binom(n, k)
2: bc =Array of size $n \times n$
3: **for all** $i = 0 \ldots n$ **do**
4: $bc[i][0] = 1$
5: $bc[i][i] = 1$
6: **for all** $i = 0 \ldots n$ **do**
7: **for all** $j = 1 \ldots i$ **do**
8: $bc[i][j] = bc[i-1][j-1] + bc[i-1][j]$
9: **return** $bc[n][k]$

Observations

- We built the table from the bottom up.
- We managed to get rid of the recursion.
- Complexity?

Observations

- We built the table from the bottom up.
- We managed to get rid of the recursion.
- Complexity? $O(n^2)$
- Much much cheaper than the brute force approach which considers a whole bunch of factorials!

Approximate String Matching

- We've looked at string matching previously.
- But what about string matching with errors?
- E.g., mispelled is closer to misspelled than mouse.
- Applications: spell checking and autocorrection, bioinformatics.
- More formally, we seek a substring "closest" to a given pattern.
- We therefore need a cost function telling us the distance between strings.

String Distance

- A reasonable distance function would capture the number of changes that must be made to convert one string to another.
- Changes:
  - substitution: replace a single character from pattern P with a character in text T, e.g., "man" $\rightarrow$ "ran".
  - insertion: insert a single character into P to match T, e.g., "par" to "part".
  - deletion: remove a single character from P to match text T, e.g., "cart" to "car".
- We can assign different costs to each of these operations, let's assume equal cost (1).
- Then the edit distance between two strings is the (minimum) number of operations required to transform from one string to another.
- How can we use this to do approximate string matching? We need to potentially consider many combinations of operations in many places in the pattern.

Observation

- Consider the last character in a string; it must be matched, substituted, inserted or deleted.
- If we remove this character from the pattern and target text, we are left with two smaller strings.
- If i and j are the last character of the prefix of pattern P and text T, then there are three pairs of shorter strings after the last operation, corresponding to match/substitution, insertion or deletion.
- If we had a cost associated with these smaller strings, we could decide which option is best, and choose appropriately (i.e., the cheapest one is most likely).
- How do we learn this cost?

Observation

- Consider the last character in a string; it must be matched, substituted, inserted or deleted.
- If we remove this character from the pattern and target text, we are left with two smaller strings.
- If i and j are the last character of the prefix of pattern P and text T, then there are three pairs of shorter strings after the last operation, corresponding to match/substitution, insertion or deletion.
- If we had a cost associated with these smaller strings, we could decide which option is best, and choose appropriately (i.e., the cheapest one is most likely).
- How do we learn this cost? recursion.

Constructing the algorithm

- Let $D[i, j]$ be the minimum number of differences between $P_1, \ldots, P_i$ and the substring of T ending at T_j. I.e., it is the minimum of the following three options.
  - If $P_i = T_j$ (i.e., the last characters are the same) then $D[i-1, j-1]$. Otherwise, $D[i, j] = D[i-1, j-1] + 1$. This option matches or substitutes the ith and jth characters, depending on whether they are the same or not.
  - $D[i-1, j] + 1$ if there is an extra character in the pattern — we have to pay the cost for an insertion.
  - $D[i, j-1] + 1$ If there is an extra character in the text we must remove.
- Computing the minimum cost requires us to recurse to every option for every character in the string; complexity is $O(3^n)$.

Dynamic Programming

- Observe that there are only $|P||T|$ unique recursive calls possible as we can only compare that many pairs. If we can store these in the table, we don't need to recompute them.
- Create a table of size $|P||T|$, with each row/column representing one character from the pattern/text.
- Each cell then contains the cost of the optimal solution to the subproblem of getting to that point, given the previous string.

	T		y	o	u	-	s	h	o	u	l	d	-	n	o	t
P	pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
:		**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	**1**	1	2	3	4	5	6	7	8	9	10	11	12	13	13
h:	2	2	**2**	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	3	**2**	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	4	3	**2**	3	4	5	6	5	6	7	8	9	10	11
-:	5	5	5	4	3	**2**	3	4	5	6	6	7	7	8	9	10
s:	6	6	6	5	4	3	**2**	3	4	5	6	7	8	8	9	10
h:	7	7	7	6	5	4	3	**2**	**3**	4	5	6	7	8	9	10
a:	8	8	8	7	6	5	4	3	3	**4**	5	6	7	8	9	10
l:	9	9	9	8	7	6	5	4	4	4	**4**	5	6	7	8	9
t:	10	10	10	9	8	7	6	5	5	5	5	**5**	6	7	8	8
-:	11	11	11	10	9	8	7	6	6	6	6	6	**5**	6	7	8
n:	12	12	12	11	10	9	8	7	7	7	7	7	6	**5**	6	7
o:	13	13	13	12	11	10	9	8	7	8	8	8	7	6	**5**	6
t:	14	14	14	13	12	11	10	9	8	8	9	9	8	7	6	**5**

Base Case and Implementation

- Note that the cost of the first row/column is its index. Why?

Base Case and Implementation

- Note that the cost of the first row/column is its index. Why?
- Because we need to match the row/column against the empty string, requiring that number of insertions or deletions.
- Each cell should also store what operation was required to reach it, in order to allow us to reconstruct these operations later. E.g., moving from "thou" to "you" requires us to do a delete, substitute, match, match.

Algorithm

```
1:  function stringCompare(s, t)
2:      initialise the cost table C                ▷ C[0][.] and C[.][0] now have costs
3:      for all i = 1...length(s) do
4:          for all j = 1...length(t) do
5:              if s[i] == t[j] then
6:                  Cost = C[i − 1][j − 1]
7:              else
8:                  Cost = C[i − 1][j − 1] + 1
9:              if C[i][j − 1] < C[i − 1][j] and C[i][j − 1] + 1 < Cost then
10:                 C[i][j] = C[i][j − 1] + 1; O[i][j] = Delete
11:             else if C[i − 1][j] < C[i][j − 1] and C[i − 1][j] + 1 < Cost then
12:                 C[i][j] = C[i − 1][j] + 1; O[i][j] = Insert
13:             else
14:                 C[i][j] = Cost
15:                 O[i][j] = Match/Substitute
16:     return C[i][j]
```

Path Reconstruction

- The algorithm returns the number of operations required to get form start to finish, and the operation at each point.
- We can move backwards from goal to start to reconstruct solution by looking at the operations.
- A substitute or match moves is from cell i, j to cell $i - 1, j - 1$ (and we can look at the characters to decide what to match/substitute.
- A deletion moves us from i, j to $i - 1, j$, and we must delete the last character from our source string.
- An insertion moves us from i, j to $i, j - 1$ and we must insert the last character in our target string.
- We then reverse these operations to finally move from source to target.

Variants

- Many classes of problems can be addressed using a similar approach.
- Often, we don't need to keep an explicit operation index, but can just move towards the cheapest neighbour to recreate solution.
- We could use different cost functions to greatly change the behaviour of the algorithm. For example, characters far apart on the keyboard could cost more to substitute.
- We've assumed the goal cell is the cell at the length of the two strings. For some variants, we might want different goals.
- We can do substring matching by changing the initial costs and goal function (to the length of the target string)
- We can find the longest common subsequence between two strings by making substitutions very expensive.

Analysis

- Algorithm has 2 nested for loops, so $O(n^2)$.
- Table is also $O(n^2)$.
- BUT, we actually only need to consider two columns at any one time - we could therefore reduce space requirements to $O(n)$. Note: we are often more limited in space than time when it comes to implementation.
- In general, to perform dynamic programming we
  1. Demonstrate the answer to a problem as a recurrence relation or via a recursive algorithm.
  2. Show that the number of different parameter values taken on by the recurrence is bounded by a polynomial.
  3. Specify how to evaluate the recurrence so that the partial results it uses are available when needed.

Longest Increasing Sequence

- We seek the longest increasing subsequence within a sequence of n numbers.

$$\{2, 4, 3, 5, 1, 7, 6, 9.8\} \rightarrow \{2, 3, 5, 6, 8\}$$

- Note that this is not a run (where elements must follow one another).
- Step 1: what recurrence can we find? What information about the first $n - 1$ elements would help us determine whether to include element n or not?
  - Length of the longest increasing sequence?

Longest Increasing Sequence

- We seek the longest increasing subsequence within a sequence of n numbers.

$$\{2, 4, 3, 5, 1, 7, 6, 9.8\} \rightarrow \{2, 3, 5, 6, 8\}$$

- Note that this is not a run (where elements must follow one another).
- Step 1: what recurrence can we find? What information about the first $n-1$ elements would help us determine whether to include element n or not?
  - Length of the longest increasing sequence? Not enough information; if we want to add the number 9 to a sequence of length 5, can we?
  - Length of the longest sequence that element n will extend.

Longest Increasing Sequence

- So the longest increasing sequence with the nth number will be formed by appending it to the longest increasing sequence to the left of n that ends on a number smaller than it.

$$l_i = max_{0<j<i} l_j + 1 \text{ such that } (s_j < s_i)$$

- The longest increasing sequence that can be found is then $max_1 \leq i \leq n l_i$
- By storing the predecessor — index of the element that should appear before it — we can start from the last value and follow predecessors to reconstruct the sequence.

Sequence s_i	2	4	3	5	1	7	6	9	8
Length l_i	1	2	3	3	1	4	4	5	5
Predecessor p_i	–	1	1	2	–	4	4	6	6