Dynamic Programming

Mark Matthews PhD

Concerned with solving problems by breaking them into (smaller) **subproblems** that are **recursively** solved.

- Subproblems are required to be **independent**

Example, Merge Sort: we solve the problem by splitting the input sequence into two **distinct** sub-sequences and recursively sorting each subsequence:

- Sort 5, 6, 2, 9, 1, 8, 3, 4 = sort 5, 6, 2, 9 and sort 1, 8, 3, 4
- Sort 5, 6, 2, 9 = Sort 5, 6 and sort 2, 9
- Sort 5, 6 = Sort 5 and Sort 6 (both sorted)
- Conquer by merging answers to subproblems.

Concerned with solving problems by breaking them into smaller subproblems that can be solved.

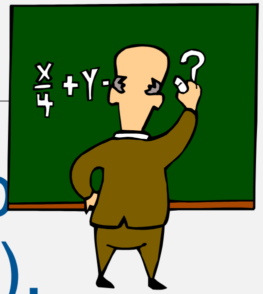
- Sub problems may be **inter-dependent**

Example, Fibonacci Numbers: $n_i = n_{i-1}+n_{i-2}$, $n_0=0$, $n_1=1$

- Solve n_5 = Solve n_4 and Solve n_3 and add the answers together
- Solve n_4 = Solve n_3 and Solve n_2 and add the answers together
- Solve n_3 = … (we are doing this twice!)

Dynamic Programming is often described as bottom-up:

- Solve the sub-problems incrementally using the answers to solve the larger problems.



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

Example: Fibonacci

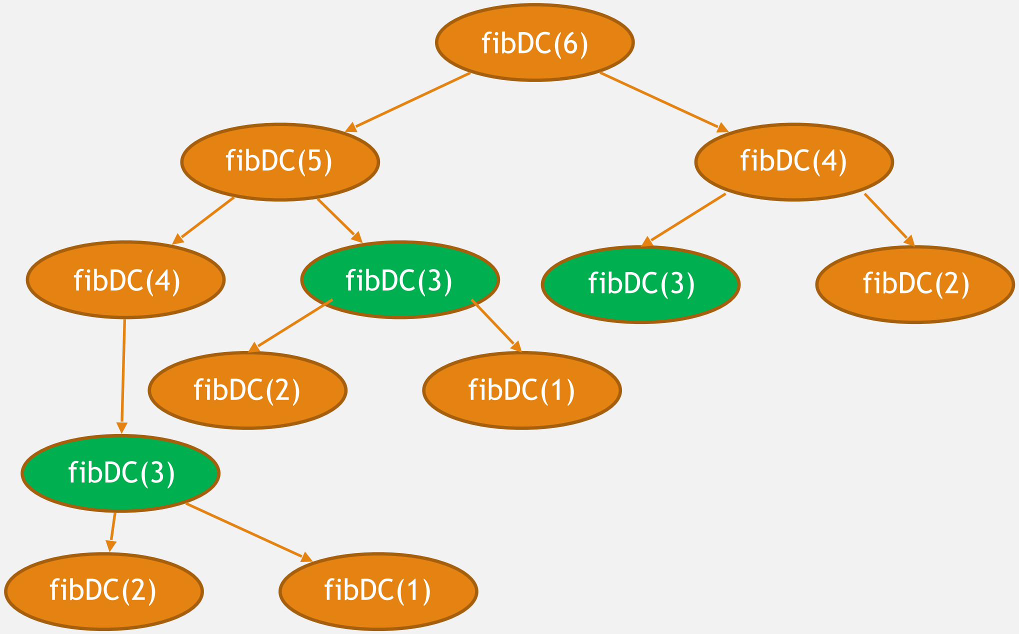
```
Algorithm: fibDC(n)
  if n < 2 then
    return n

  return fib(n-1) + fib(n-2)
```

Example: Fibonacci

```
Algorithm: fibDC(n)
  if n < 2 then
    return n

  return fib(n-1) + fib(n-2)
```



Example: Fibonacci

```
Algorithm: fibDC(n)
  if n < 2 then
    return n


  return fib(n-1) + fib(n-2)
```

```
Algorithm: fibDP(n)
  F = array size n
  index := 0

  F[0] = 0
  if (n > 0) then
    F[1] = 1
    for i=2 to n do
      F[i] = F[i-1]+F[i-2]

  return F[n]
```

Example: Fibonacci

```
Algorithm: fibDC(n)

  if n < 2 then

    return n


  return fib(n-1) + fib(n-2)
```

```
Algorithm: fibDP(n)
  F = array size n
  index := 0

  F[0] = 0
  if (n > 0) then
    F[1] = 1
    for i=2 to n do
      F[i] = F[i-1]+F[i-2]

  return F[n]
```

i	0	1	2	3	4	5	6
F[i]	0	1	1	2	3	5	8

Example: Fibonacci

```
Algorithm: fibDC(n)

  if n < 2 then

    return n


  return fib(n-1) + fib(n-2)
```

```
Algorithm: fibDP(n)
  F = array size n
  index := 0

  F[0] = 0
  if (n > 0) then
    F[1] = 1
    for i=2 to n do
      F[i] = F[i-1]+F[i-2]

  return F[n]
```

i	0	1	2	3	4	5	6
F[i]	0	1	1	2	3	5	8

Dynamic Programming = Solve the problem incrementally, using a **bottom-up** approach storing intermediary answers in a **table**

Many practical applications of dynamic programming involve selecting from amongst a set of possible answers.

- Shortest path = there are many paths between two points, which is the shortest?
- Longest Common Subsequence between two strings?

Selecting one answer from many possibilities is an optimisation problem:

- There is a recursive relationship between a problem and an associated subproblem.
- There are (potentially) many valid optimal solutions
- **Principle of Optimality:** The optimal solution to the problem can be built from the optimal solutions to the subproblems

4 Steps of Dynamic Programming:

1. **Optimal Substructure:** Define subproblems and d characterise the structure of an optimal solution.
2. **Recursive Formulation:** Recursively define the value of an optimal solution.
3. **Recurse&Memoize/Tabulate**
   - **Top-down:** Build a recursive solutions that uses a memory to remember previous calculations.
   - **Bottom-up:** Incrementally construct the solution using a table that contains solutions to subproblems.
4. **Construct and Optimal Solution** from the computed information.

Subsequences

- A **subsequence** of a character string $x_0x_1x_2\ldots x_{n-1}$ is a string of the form $x_{i1}x_{i2}\ldots x_{ik}$, where ij < ij+1.
- Not the same as substring!
- Example String: ABCDEFGHIJK
  - Subsequence: ACEGJIK
  - Subsequence: DFGHK
  - Not subsequence: DAGH

Longest Common Subsequence Problem

Given: Two sequences

- $X = <x_1, x_2, \ldots, x_m> = X[1\ldots m]$
- $Y = <y_1, y_2, \ldots, y_n> = Y[1\ldots n]$

Find: a longest subsequence common to both X and Y

- HUMAN & CHIMPANZEE = AN
- TERMINATOR & THERMOMETER = ERM

Practical Applications:

- Identifying conserved regions in two protein or DNA sequences
- UNIX diff utility

A Poor Approach to the LCS Problem

- A Brute-force solution:
  - Enumerate all subsequences of X
  - Test which ones are also subsequences of Y
  - Pick the longest one.
- Analysis:
  - If X is of length n, then it has 2^n subsequences
  - This is an exponential-time algorithm!

For all sub-sequences S_X of X, check if S_X is also a subsequence of Y.

- Assuming the length of Y is n and the length of S_X is O(m) [but <n]
- Finding S_X in Y takes O(m+n) time (KMP / Boyer-Moore)

But:

- There are 2^m possible sub-sequences, S_X in X (powerset of X).
- Therefore – the algorithm runs in **$O(2^m(m+n))$** time

Can we do better with Dynamic Programming?

DP Approach

Consider two strings X and Y:

- $X = <x_1, .., x_m>$ and $Y = <y_1, \ldots, y_m>$

Let X_i and Y_j be prefixes of X and Y respectively

- $X_i = <x_1, .., x_i>$ for all $0 < i < m$
- $Y_j = <y_1, \ldots, y_j>$ for all $0 < j < n$

We can define the length of longest common subsequence of X_i and Y_j, denoted LCS[i, j] for all values of i and j.

LCS can be represented as a table with X representing the rows and Y representing the columns.

The values for a given row and column are the length of the longest common subsequence of the prefixes of X and Y,

DP Approach

We can define LCS[i, j] recursively using 2 cases:

Case 1: $x_i = y_j$

- We have a match, so the value depends on the previous value
- LCS[i, j] = 1 + LCS[i-1, j-1]

Case 2: $x_i \mathrel{!=} y_j$

- We have a mismatch, so the character cannot contribute to the LCS.
- LCS was based on one of the substrings of X_i and Y_j
- LCS[i, j] = max(LCS[i-1, j], LCS[i, j-1])

DP Approach

To solve a problem, we construct the table LCS[m,n]

We explore the table, following the gradient from the highest value to the lowest value, which identifies the longest common subsequence.

	C	H	I	M	P	A	N	Z	E	E
H	0	1	1	1	1	1	1	1	1	1
U	0	1	1	1	1	1	1	1	1	1
M	0	1	1	2	2	2	2	2	2	2
A	0	1	1	2	2	3	3	3	3	3
N	0	1	1	2	2	3	4	4	4	4

Analysis of LCS Algorithm

- We have two nested loops
  - The outer one iterates n times
  - The inner one iterates m times
  - A constant amount of work is done inside each iteration of the inner loop
  - Thus, the total running time is $O(nm)$
- Answer is contained in $L[n,m]$ (and the subsequence can be recovered from the L table).

Java Implementation

```java
/** Returns table such that L[j][k] is length of LCS for X[0..j-1] and Y[0..k-1]. */
public static int[ ][ ] LCS(char[ ] X, char[ ] Y) {
  int n = X.length;
  int m = Y.length;
  int[ ][ ] L = new int[n+1][m+1];
  for (int j=0; j < n; j++)
    for (int k=0; k < m; k++)
      if (X[j] == Y[k])          // align this match
        L[j+1][k+1] = L[j][k] + 1;
      else                       // choose to ignore one character
        L[j+1][k+1] = Math.max(L[j][k+1], L[j+1][k]);
  return L;
}
```

Java Implementation, Output of the Solution

```
/** Returns the longest common substring of X and Y, given LCS table L. */
public static char[ ] reconstructLCS(char[ ] X, char[ ] Y, int[ ][ ] L) {
  StringBuilder solution = new StringBuilder( );
  int j = X.length;
  int k = Y.length;
  while (L[j][k] > 0)                          // common characters remain
    if (X[j−1] == Y[k−1]) {
      solution.append(X[j−1]);
      j−−;
      k−−;
    } else if (L[j−1][k] >= L[j][k−1])
      j−−;
    else
      k−−;
  // return left-to-right version, as char array
  return solution.reverse( ).toString( ).toCharArray( );
}
```

Dynamic Programming Summary

- If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor
- No overhead for recursion and less overhead for maintaining table
- There are some problems for which the regular pattern of table accesses in the dynamic-programming algorithm can be exploited to reduce the time or space requirements even further
- If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required