Dynamic Programming

Dr Timothy Kimber

March 2015

Another Scheme

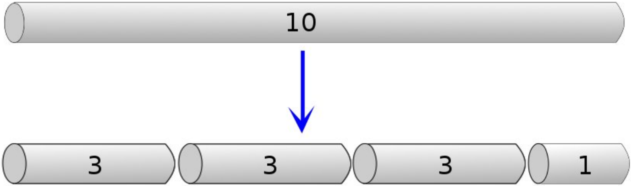
Given a problem involving choices

- Divide and Conquer may not be appropriate
- A greedy algorithm might not be possible

An Example Problem

The Rod Cutting Problem is a classic example

- A business buys steel rods in a variety of lengths
- They will cut the rods into smaller pieces to sell on
- Each rod size has a different market value
- What is the maximum profit for a rod of length N?

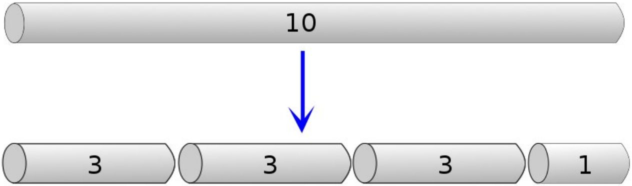


Is $p(3) + p(3) + p(3) + p(1) > p(4) + p(4) + p(2)$?

An Example Problem

If the selling prices for each size of rod up to 10 are

size	1	2	3	4	5	6	7	8	9	10
price	3	4	6	9	16	20	22	24	26	30



Then the answer for $N = 10$ is 32 ($1 \times 6 + 4 \times 1$, or 2×5)

Rod Cutting Problem

Problem *(Rod Cutting)*

Input: an array P of numbers $[P_1, \ldots, P_k]$.

Input: an integer N between 1 and k.

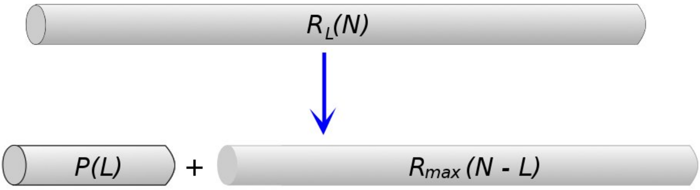
Output: a number R such that there are integers $\langle s_1, \ldots, s_j \rangle$, and $N = \sum_{i=1}^{j} s_i$, and $R = \sum_{i=1}^{j} P[s_i]$ and there is no R', defined equivalently, where $R' > R$.

- N is the length of rod to be cut up
- $P[i]$ is the selling price for a piece of size i
- $\langle s_1, \ldots, s_j \rangle$ are the sizes of the cut pieces
- R is the maximum revenue from a rod of size N
- We might want to know $\langle s_1, \ldots, s_j \rangle$ but will focus on R

The First Cut

If we make a cut at L we now have two pieces

- Keep the first piece (this was the choice we made)
- Continue cutting up the other piece
- So, the total revenue is $R_L(N) = P[L] + R_{max}(N - L)$



- Each L will give a different $R_L(N)$
- Need to compute all $R_i(N)$ and find the maximum

A Simple Recursive Solution

SimpleRodCut(Input: N, $P = [P_1, \ldots, P_k]$)

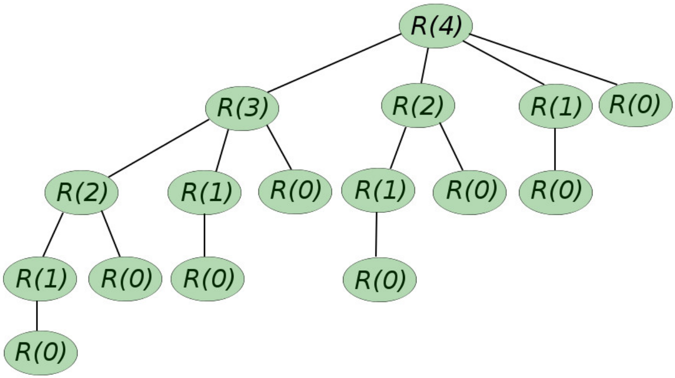
- If $N == 0$
  - Return 0
- Else
  - For $i = 1$ to N
    - $Choices[i] = P[i] + SimpleRodCut(N - i, P)$
  - Return $Max(Choices)$

- *Choices* collects totals for each possible first cut
- *Max* finds the maximum of the choices

How does this run?

Simple Rod Cut — Reflection

WOW that was sloooooowww. What happened?



- Computation for $R(N)$ is more than twice the size of $R(N-1)$

Performance

The running time of SimpleRodCut is

$$T(0) = \Theta(1)$$
$$T(N) = \Theta(1) + \sum_{i=0}^{N-1} T(i) \qquad , \text{for } N > 0$$

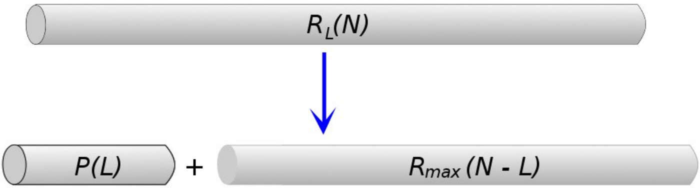
which gives $T(N) = \Theta(2^N)$.

- The running time grows exponentially.
- This is not a practical solution.

Greedy?

- The problem involves making choices (where is the first cut?)
- What greedy choices could be made?

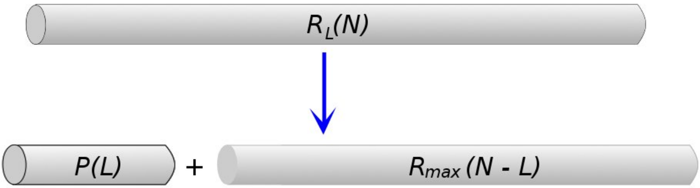
size	1	2	3	4	5	6	7	8	9	10
price	3	4	6	9	16	20	22	24	26	30



Greedy?

- The problem involves making choices (where is the first cut?)
- What greedy choices could be made?

size	1	2	3	4	5	6	7	8	9	10
price	3	4	6	9	16	20	22	24	26	30

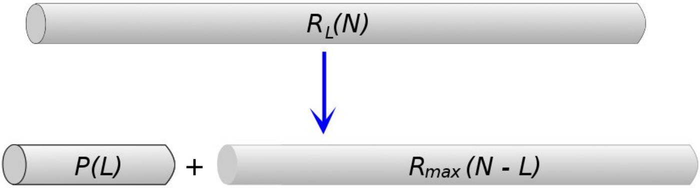


EXERCISE: Show these greedy choices are not correct.

Divide & Conquer?

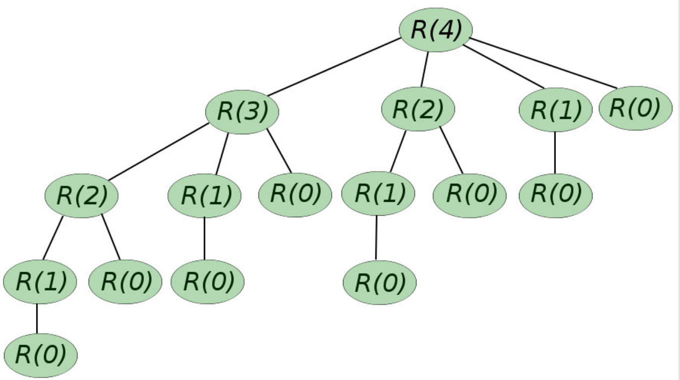
- Can we divide the problem? (and conquer?)

size	1	2	3	4	5	6	7	8	9	10
price	3	4	6	9	16	20	22	24	26	30



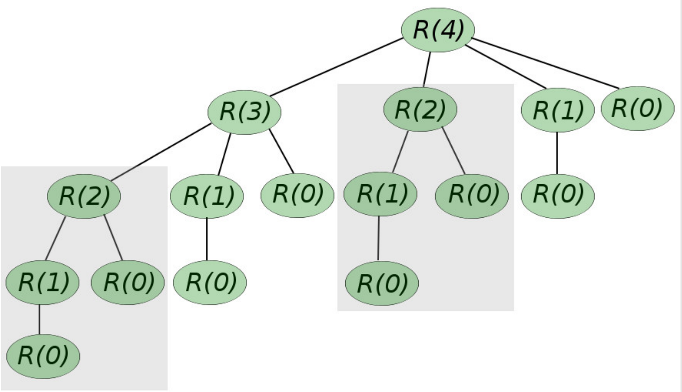
New Strategy

What is there that we can take advantage of?



New Strategy

What is there that we can take advantage of?



- The subproblems overlap

Dynamic Programming

Dynamic Programming makes a space–time tradeoff

- Do not want to recompute the answer to $R(i)$ every time
- Compute it once and save the answer in a table
- Check the table before computing each subproblem

This is called memoisation (we are making a note for later)

MemoisedRodCut(Input: N, $P = [P_1, \ldots, P_k]$)

- For $i = 0$ to N
  - $R[i] = 0$
- Return $MemoiseAux(N, P, R)$

- R is the table to be filled in

Memoisation

MemoiseAux(Input: N, $P = [P_1, \ldots, P_k]$, $R = [R_0, \ldots, R_{N'}]$)

- If $N == 0$
  - Return 0
- If $R[N] > 0$
  - Return $R[N]$
- For $i = 1$ to N
  - $Choices[i] = P[i] + MemoiseAux(N - i, P, R)$
- $R[N] = Max(Choices)$
- Return $R[N]$

- If $R[N]$ was already computed ($R[N] > 0$) it is returned immediately
- Otherwise we compute it, save it, and then return it
- This approach is also called Top Down

The 'Bottom Up' Method

We know which problems depend on which others

- so we can just complete the table in order
- this will be more efficient than recursion

BottomUpRodCut(Input: N, $P = [P_1, \ldots, P_k]$)

- $R[0] = 0$
- For $i = 1$ to N
  - $Choices = [0, \ldots, 0]$
  - For $j = 1$ to i
    - $Choices[j] = P[j] + R[i - j]$
  - $R[i] = Max(Choices)$
- Return $R[N]$

- What is the running time?

Dynamic Programming

Dynamic programming can be applied to a problem if

- The problem has optimal substructure
- The problem has overlapping subproblems

A problem has optimal substructure if

- the problem can be decomposed into subproblems
- an optimal solution uses optimal solutions to the subproblems

In rod cutting the optimal solution for N was one of

- $P[i] + R[N-i]$, where $1 \leq i < N$

and each $R[N-i]$ was an optimal solution for $N-i$.

Optimal Substructure

Problems may appear to have optimal substructure when they do not

Problem *(Unweighted Shortest Path)*

Input: graph $G = (V, E)$.

Input: vertices $u, v \in V$.

Output: the simple path from u to v containing the fewest edges

Problem *(Unweighted Longest Path)*

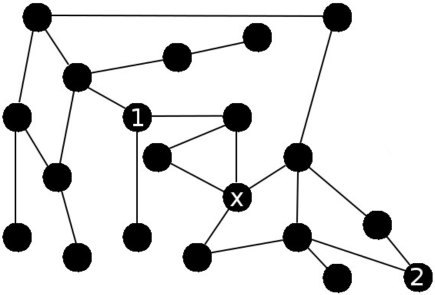
Input: graph $G = (V, E)$.

Input: vertices $u, v \in V$.

Output: the simple path from u to v containing the most edges

Optimal Substructure

A shortest path is composed of optimal solutions to subproblems

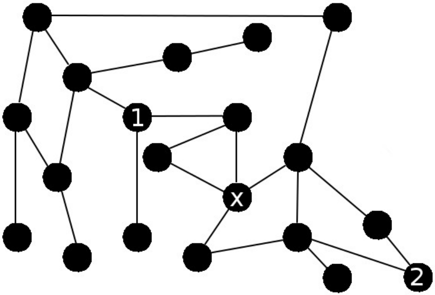


The shortest path from 1 to 2 (via x) is

- shortest path from 1 to x
- plus the shortest path from x to 2

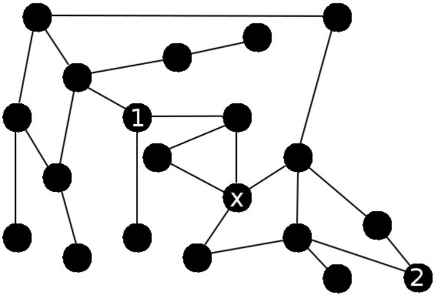
Optimal Substructure

How about a longest path?



Optimal Substructure

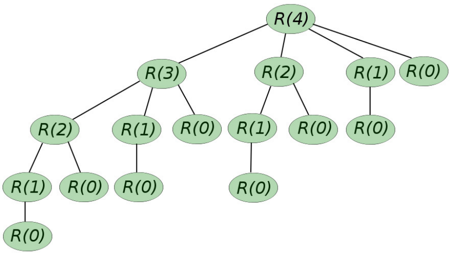
How about a longest path?



- Independent subproblem solutions do not make an optimal solution
- In an optimal solution the subproblems will interfere

Overlapping Subproblems

The second property we need when applying dynamic programming is overlapping subproblems

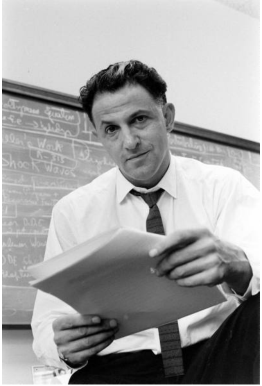


- The same problems are generated over and over
- The subproblems must still be independent
- The set of all subproblems is the subproblem space
- The smaller the subproblem space the quicker the (dynamic) algorithm

Why 'Dynamic Programming'?

Dynamic programming was developed by Richard Bellman (of Bellman–Ford fame) in the 1950s.

- He was working (indirectly) for the US govt
- His boss didn't like research
- The name was chosen to make his work sound less theoretical
- *Programming* actually refers to the generation of a 'program' or plan
- *Dynamic* hints at time dependence but mostly just sounds good



So, not the most descriptive term ever, but it stuck.

Another Example

The longest common subsequence is a way of quantifying the similarity between two strings.

A G C G A T A T C C A C T G

T C A C G C A T A G G A C T

- A subsequence is any sequence contained within the string
- It does not need to be continuous
- A string of length N has 2^N subsequences
- So, a brute force method would take $O(2^N)$ time

Another Example

The longest common subsequence is a way of quantifying the similarity between two strings.

A G C G A T A T C C A C T G

T C A C G C A T A G G A C T

- A subsequence is any sequence contained within the string
- It does not need to be continuous
- A string of length N has 2^N subsequences
- So, a brute force method would take $O(2^N)$ time

LCS Problem

Definition (Subsequence)

Let $T = \langle T_1, \ldots, T_N \rangle$ be a sequence. A sequence $S = \langle S_1, \ldots, S_M \rangle$ is a *subsequence* of T if and only if there is a strictly increasing sequence $\langle i_1, \ldots, i_M \rangle$ of indices of T such that $S_j = T_{i_j}$ for all j in $1, \ldots, M$.

Problem *(Longest Common Subsequence)*

Input: strings $X = [X_1, \ldots, X_M]$ and $Y = [Y_1, \ldots, Y_N]$.

Output: an integer L such that there is a string $Z = [Z_1, \ldots, Z_L]$ that is a subsequence of both X and Y.

- M is the length of X, N is the length of Y
- Z is the common subsequence, L is the length of Z
- Focus on L for now

Subproblems

Consider two strings X and Y

A G C G A T A T C C A C T G
T C A C G C A T A G G A C G

If X and Y end with the same character

- The character must be in longest common subsequence Z
- The subproblem to solve is to find the LCS of $[X_1, \ldots, X_{M-1}]$ and $[Y_1, \ldots, Y_{N-1}]$
- L is $1 +$ length of subproblem LCS

Subproblems

Consider two strings X and Y

A G C G A T A T C C A C T G
T C A C G C A T A G G A C T

Otherwise, if X_M and Y_N are different

- They cannot both be in Z, but either might be
- We have two subproblems
  - find the LCS of X and $[Y_1, \ldots, Y_{N-1}]$
  - find the LCS of $[X_1, \ldots, X_{M-1}]$ and Y
- L is the length of the longer of the two

Does the problem have optimal substructure?

Subproblems

Consider two strings X and Y

A G C G A T A T C C A C T G
T C A C G C A T A G G A C T

Otherwise, if X_M and Y_N are different

- They cannot both be in Z, but either might be
- We have two subproblems
  - find the LCS of X and $[Y_1, \ldots, Y_{N-1}]$
  - find the LCS of $[X_1, \ldots, X_{M-1}]$ and Y
- L is the length of the longer of the two

Does the problem have optimal substructure?

Subproblems

Consider two strings X and Y

A G C G A T A T C C A C T G
T C A C G C A T A G G A C T

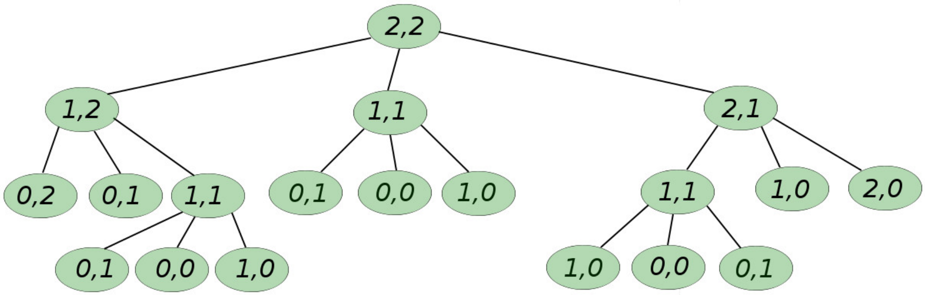
Otherwise, if X_M and Y_N are different

- They cannot both be in Z, but either might be
- We have two subproblems
  - find the LCS of X and $[Y_1, \ldots, Y_{N-1}]$
  - find the LCS of $[X_1, \ldots, X_{M-1}]$ and Y
- L is the length of the longer of the two

Does the problem have optimal substructure?

Subproblem Graph

The overall subproblem graph has this form



- Any problem involving an empty string is a base case

Subproblem Space

The space of subproblems has size $(M + 1) \times (N + 1)$

	0	G 1	G 2	T 3	A 4	T 5
0	0	0	0	0	0	0
C 1	0	0	0	0	0	0
G 2	0	1	1	1	1	1
A 3	0	1	1	1	2	2
C 4	0	1	1	1	2	2
T 5	0	1	1	2	2	3

- Given strings will not generate all the subproblems
- What are the best and worst cases?

Dynamic Programming

It appears that a dynamic programming solution is applicable

- The problem has optimal substructure
- There are overlapping subproblems
- This should give $\Theta(M \times N)$ performance

	0	G 1	G 2	T 3	A 4	T 5
0	0	0	0	0	0	0
C 1	0	0	0	0	0	0
G 2	0	1	1	1	1	1
A 3	0	1	1	1	2	2
C 4	0	1	1	1	2	2
T 5	0	1	1	2	2	3

Java Implementation

This is the base class for an LCS computation

```java
public abstract class AbstractLCS {
    protected char[] x;
    protected char[] y;
    protected int[][] s;

    public AbstractLCS(char[] x, char[] y) {
        this.x = x;
        this.y = y;
    }

    public int length() {
        return s[x.length][y.length];
    }

```

- Two arrays of chars x and y are supplied to the class
- The computation occurs in subclasses
- The size of each subproblem LCS is saved in array s

Bottom-Up Computation

```java
public void findLCS() {
    int m = x.length;
    int n = y.length;
    s = new int[m + 1][n + 1];
    for (int i = 1; i <= m; i++) {
        Arrays.fill(s[i], -1);
        s[i][0] = 0;
    }

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (x[i - 1] == y[j - 1]) { s[i][j] = 1 + s[i-1][j-1]; }
            else {
                s[i][j] = Math.max(s[i][j-1], s[i-1][j]);
            }
        }
    }
}
```

- The default value for an `int[]` is 0
- It fills s row by row

Top-Down Computation

```java
public void findLCS() {
    int m = x.length;
    int n = y.length;
    s = new int[m + 1][n + 1];
    for (int i = 1; i <= m; i++) {
        Arrays.fill(s[i], -1);
        s[i][0] = 0;
    }
    findLCSAux(m, n);
}

private void findLCSAux(int m, int n) {
    if (s[m][n] >= 0)         { return; }
    if (x[m - 1] == y[n - 1]) {
        findLCSAux(m - 1, n - 1);
        s[m][n] = 1 + s[m - 1][n - 1];
    } else {
        findLCSAux(m - 1, n);
        findLCSAux(m, n - 1);
        s[m][n] = Math.max(s[m - 1][n], s[m][n - 1]);
    }
}
```

Constructing the LCS

The LCS string can be recreated by looking at the contents of s

```
public char[] lcs() {
    int l = length();
    char[] lcs = new char[l];
    writeLCS(x.length, y.length, lcs, l - 1);
    return lcs;
}

private void writeLCS(int m, int n, char[] lcs, int i) {
    if      (m == 0 || n == 0)       { return; }
    if      (s[m][n] == s[m - 1][n]) { writeLCS(m - 1, n, lcs, i); }
    else if (s[m][n] == s[m][n - 1]) { writeLCS(m, n - 1, lcs, i); }
    else {
        writeLCS(m - 1, n - 1, lcs, i - 1);
        lcs[i] = x[m - 1];
    }
}

public String toString() {
    return new String(lcs());
}
```

Performance

The brute force approach would enumerate all subsequences of one string

- This produces exponential performance
- Recomputing all subproblems has the same problem

The performance of either dynamic programming solution is $O(M \times N)$

- Either an addition or a comparison for each subproblem

For best case input the top-down approach has $\Theta(N)$ performance

- Only the necessary subproblems are computed
- Bottom-up always computes everything
- But in this case the brute force solution is also $\Theta(N)$