Maximum Subarray Sum

The Problem

Given an integer array A (which may contain negative numbers), find the contiguous subarray with the largest sum, and return that sum.

Note: the subarray must be **contiguous** and non-empty.

Example: A = [−2, 1, −3, 4, −1, 2, 1, −5, 4] → 6 (subarray [4, −1, 2, 1])

Any ideas on how this can be solved using dynamic programming?

Hints

- Define `M[i]` as the best sum of a subarray that *ends exactly at* index i.
- For the subarray ending at i: either extend the best one ending at i−1, or start fresh at A[i]. When is starting fresh better?
- The best subarray does not have to end at the last index. Where could it end?

The Solution

1. **Subproblem:**
   `M[i]` = maximum sum of a subarray ending exactly at index i.
2. **Base Case:**
   `M[0] = A[0]`
3. **Recurrence:**
   `M[i] = max(A[i], M[i−1] + A[i])`
   - **Start fresh:** `A[i]` (when M[i−1] < 0)
   - **Extend:** `M[i−1] + A[i]`
4. **Final Answer:**
   The maximum of M[i] over all i, tracked in a variable best.

Example

Input: A = [−2, 1, −3, 4, −1, 2, 1, −5, 4]

i	0	1	2	3	4	5	6	7	8
A[i]	-2	1	-3	**4**	**-1**	**2**	**1**	-5	4
M[i]	-2	1	-2	4	3	5	**6**	1	5
best	-2	1	1	4	4	5	6	6	6

```
M[2] = max(−3, 1 − 3) = −2
M[3] = max(4, −2 + 4) = 4   ← start fresh
M[6] = max(1, 5 + 1) = 6   ← largest
```

Output: 6
Subarray [4, −1, 2, 1] (indices 3 to 6)

Complexity Analysis

1. **Number of Subproblems:**
   - One subproblem M[i] per index, so there are n subproblems.
2. **Work per Subproblem:**
   - One addition and two max operations, which is O(1).
3. **Time Complexity:**
   - Total = n × O(1) = O(n), a single pass.
   - Compare with brute force over all $O(n^2)$ subarrays: $O(n^2)$ with running sums, $O(n^3)$ without.

Complexity Analysis (contd.)

1. **Space Complexity:**
   - The array M[0..n−1] gives O(n) space.
2. **Space Optimization (Kadane's Algorithm):**
   - M[i] only needs M[i−1], so keeping one variable (current) plus best gives O(1) space.

Minimum Cost Path in a Grid

The Problem

Given an m × n grid filled with non-negative numbers, find a path from the top-left corner (0, 0) to the bottom-right corner (m−1, n−1) that minimizes the sum of all numbers along its path.

Note: you can only move either **down** or **right** at any point in time.

Example: for the grid [[1, 3, 1], [1, 5, 1], [4, 2, 1]], the answer is 7.

Any ideas on how this can be solved using dynamic programming?

Hints

- Look at any cell (i, j). What are the only two cells you could have come from?
- How does the minimum cost to reach (i, j) depend on the minimum cost to reach its top neighbour (i−1, j) and its left neighbour (i, j−1)?
- What are the boundary conditions for the first row and first column, where movement is restricted to only one direction?

The Solution

1. **Subproblem:**
   `C[i, j]` = minimum cost of any path from (0, 0) to cell (i, j).
2. **Base Cases:**
   - `C[0, 0] = grid[0, 0]`
   - **First row:** `C[0, j] = C[0, j−1] + grid[0, j]` (can only come from the left)
   - **First column:** `C[i, 0] = C[i−1, 0] + grid[i, 0]` (can only come from above)
3. **Recurrence:**
   ```
   C[i, j] = grid[i, j] + min(C[i−1, j], C[i, j−1])
   ```
4. **Order of Computation:**
   Fill the table row by row, left to right, so both neighbours are ready before (i, j). The answer is C[m−1, n−1].

Example

Input grid

1	3	1
1	5	1
4	2	1

DP table C

1	4	5
2	7	6
6	8	7

Base cases

```
Row 0: [1, 4, 5]
Col 0: [1, 2, 6]
```

Fill the table

```
C[1,1] = 5 + min(4, 2) = 7
C[1,2] = 1 + min(5, 7) = 6
C[2,1] = 2 + min(7, 6) = 8
C[2,2] = 1 + min(6, 8) = 7
```

Output: 7

Path: 1 → 3 → 1 → 1 → 1 (highlighted)

Complexity Analysis

1. **Number of Subproblems:**
   - One subproblem C[i, j] per cell, so there are m × n subproblems.
2. **Work per Subproblem:**
   - Each cell looks at two neighbours and takes a min, which is O(1).
3. **Time Complexity:**
   - Total = (m × n subproblems) × O(1) = O(m × n).
   - Compare with plain recursion, which explores every path: O(2^(m+n)).

Complexity Analysis (contd.)

1. **Space Complexity:**
   - The DP table C stores one value per cell, so space is O(m × n).
2. **Space Optimization:**
   - Row i only needs row i−1 (above) and the current row (left).
   - Keeping a single 1D array of size n and updating it in place reduces space to O(n).

Counting Unique BSTs

The Problem

Given an integer n, count how many structurally different binary search trees (BSTs) can be built using the keys 1, 2, ..., n.

Recall: in a BST, every key in the left subtree is smaller than the root and every key in the right subtree is larger.

Example: for n = 3, there are 5 unique BSTs.

Any ideas on how this can be solved using dynamic programming?

Hints

- Pick a key r ($1 \leq r \leq n$) to be the root. Which keys must go into its left subtree, and which into its right subtree?
- With r at the root, the left subtree is a BST on $r-1$ keys and the right subtree is a BST on $n-r$ keys. How do their counts combine?
- Does the count depend on the actual key values, or only on how many keys there are? How many BSTs are there with 0 keys? With 1 key?

The Solution

1. **Subproblem:**
   `T[m]` = number of unique BSTs that can be built from m keys.
2. **Base Cases:**
   - `T[0] = 1` (the empty tree)
   - `T[1] = 1` (a single node)
3. **Recurrence:**
   `T[m] = Σ (r = 1 to m)  T[r−1] × T[m−r]`
   - **Root r:** `T[r−1]` choices for the left subtree × `T[m−r]` for the right
   - Different roots give different trees, so add over all r.
4. **Order of Computation:**
   Compute T[2], T[3], …, T[n] in order, since each needs all smaller values. The answer is T[n].

Example

Input: n = 4

m	0	1	2	3	4
T[m]	1	1	2	5	14

```
T[0] = 1,  T[1] = 1
T[2] = T0·T1 + T1·T0
     = 1 + 1 = 2
T[3] = T0·T2 + T1·T1 + T2·T0
     = 2 + 1 + 2 = 5
T[4] = T0·T3 + T1·T2 + T2·T1 + T3·T0
     = 5 + 2 + 2 + 5 = 14
```

Output: 14
Reading T[4]: root 1 → 0 keys left, 3 right (1 × 5);
root 2 → 1 left, 2 right (1 × 2); root 3 → 2 left,
1 right (2 × 1); root 4 → 3 left, 0 right (5 × 1).
These are the Catalan numbers: 1, 1, 2, 5, 14, 42, …

Complexity Analysis

1. **Number of Subproblems:**
   - One subproblem T[m] for each m = 0 … n, so there are n + 1 subproblems.
2. **Work per Subproblem:**
   - T[m] sums over m possible roots, which is O(m) work.
3. **Time Complexity:**
   - Total = 1 + 2 + … + n = n(n+1)/2 = $O(n^2)$.
   - Plain recursion recomputes the same T[k] again and again, which takes exponential time.

Complexity Analysis (contd.)

1. **Space Complexity:**
   - The array T[0..n] gives O(n) space.
   - All smaller values are needed for each T[m], so this cannot be reduced the way the other problems can.
2. **Connection:**
   - T[n] is the n-th Catalan number, with closed form C(2n, n) / (n + 1).
   - The same numbers count valid parenthesizations, which links back to Matrix Chain Multiplication.

Longest Palindromic Subsequence

The Problem

Given a string s, find the length of its longest palindromic subsequence.

A **subsequence** is derived from another sequence by deleting zero or more elements without changing the order of the remaining elements. Unlike a substring, it does not need to be contiguous.

Example: for s = "bbbab", the answer is 4 ("bbbb").

Any ideas on how this can be solved using dynamic programming?

Hints

- Look at the boundary characters of a substring s[i…j]. If s[i] == s[j], how much do they contribute to the palindrome length?
- If they do not match, explore two choices: drop the left character (s[i+1…j]) or drop the right character (s[i…j−1]).
- In what order should the subproblems be solved? Think about solving by substring length.

The Solution

1. **Subproblem:**
   `L[i, j]` = length of the longest palindromic subsequence of s[i…j].
2. **Base Cases:**
   - `L[i, i] = 1` (every single character is a palindrome)
   - **Length 2 with s[i] == s[j]:** `L[i, j] = 2` (the inner substring is empty, length 0)
3. **Recurrence:**
   - **If s[i] == s[j]:** `L[i, j] = L[i+1, j−1] + 2`
   - **If s[i] ≠ s[j]:** `L[i, j] = max(L[i+1, j], L[i, j−1])`
4. **Order of Computation:**
   Solve by increasing substring length (1, 2, …, n), so shorter substrings are ready first. The answer is L[0, n−1].

Example

s = "bbbab" → table L[i, j]

i/j	0 b	1 b	2 b	3 a	4 b
0 b	1	2	3	3	4
1 b		1	2	2	3
2 b			1	1	3
3 a				1	1
4 b					1

Length 1: every L[i, i] = 1
Length 2: "bb" → 2, "ba" → 1, "ab" → 1
Length 3:
`"bbb": b = b → L[1,1] + 2 = 3`
`"bba": b ≠ a → max(1, 2) = 2`
`"bab": b = b → L[3,3] + 2 = 3`
Length 4: "bbba" → 3, "bbab" → 3
Length 5:
`"bbbab": b = b → L[1,3] + 2 = 4`

Output: 4 (longest palindromic subsequence: "bbbb")

Complexity Analysis

1. **Number of Subproblems:**
   - One subproblem L[i, j] for every pair $i \leq j$, which is about $n^2/2 = O(n^2)$.
2. **Work per Subproblem:**
   - One character comparison plus a lookup or a max of two values, which is O(1).
3. **Time Complexity:**
   - Total = $O(n^2)$ subproblems × O(1) = $O(n^2)$.
   - Compare with checking every subsequence: there are 2^n of them.

Complexity Analysis (contd.)

1. **Space Complexity:**
   - The n × n DP table L gives $O(n^2)$ space.
2. **Space Optimization:**
   - Row i only depends on row i+1, so two 1D arrays of size n are enough, which gives O(n) space.
3. **Connection:**
   - LPS(s) = LCS(s, reverse(s)), the Longest Common Subsequence of s and its reverse.
4. **Contrast with Maximum Subarray:**
   - Maximum Subarray needed a contiguous block; LPS works with subsequences, where gaps are allowed.