Max Subsequence Sum Problem: (MSS)

**Input:** A sequence \(x_1, x_2, \ldots, x_n\) of integers. (Some of the integers may be negative.)

**Output:** The maximum sum over all subsequences.

**Initial attempt:**
- No. of subsequences is \(O(n^2)\).
- The sum of any subsequence can be found in \(O(n)\) time.
- Thus, the sums of all the subsequences can be found in \(O(n^3)\) time.
- Leads to an \(O(n^3)\) algorithm for MSS.

**A simple improvement:**
- Let \(S[i, j]\) denote the sum of the subsequence \(x_i, x_{i+1}, \ldots, x_j\) (\(1 \leq i \leq j \leq n\)).
- Fact: \(S[i, j + 1] = S[i, j] + x_{j+1}\). So, given \(S[i, j]\), \(S[i, j + 1]\) can be computed in \(O(1)\) time.
- Thus, all the values \(S[i, i], S[i, i + 1], \ldots, S[i, n]\) can be computed in \(O(n)\) time.
- Thus, the sums of all the subsequences can be found in \(O(n^2)\) time.
- Leads to an \(O(n^2)\) algorithm for MSS.

**A D-and-C algorithm for MSS:** Sequence is stored in \(A[1 \ldots n]\).

**Divide step:** Divide the array into two equal halves.

**Conquer step:** Recursively compute the max subsequence sums of the left and right halves (\(s_\ell\) and \(s_r\) respectively) of the array.

Recursion stops when the subarray is of size 1. (In that case, the maximum sum is the value of the array element.)

**Combine step:**

\[
A = \begin{array}{c|c|c|c}
1 & \ldots & n/2 \backslash n/2+1 \ldots n \\
\hline
\text{Left Half} & \text{||} & \text{Right Half}
\end{array}
\]
(a) Max sum subsequence may lie entirely in the left half or entirely in the right half.

(b) Max sum subsequence may straddle the two halves.

Possibility (a): The candidate solution values are \(s_\ell\) and \(s_r\).

Possibility (b): Must find the max sum among all the subsequences straddling the two halves.

Observation: Every max sum subsequence that straddles the two halves must contain both \(A[n/2]\) and \(A[n/2 + 1]\). So,

1. Find the max sum (say \(s_e\)) among all subsequences ending at \(x_{n/2}\).
2. Find the max sum (say \(s_b\)) among all subsequences beginning at \(x_{n/2+1}\).
3. \(s_m = s_e + s_b\) is the maximum sum due to a straddling subsequence.

Answer from the combine step = \(\max\{s_\ell, s_r, s_m\}\).

Pseudocode: Simple exercise.

Running time analysis:

- Assume \(n = 2^r\) (\(r\) : non-negative integer).
- \(T(n)\) : Running time for array of size \(n\).
- Divide time = \(c_1\) (constant).
- Conquer time = \(2T(n/2)\).
- Combine time = \(O(n)\); that is, \(\leq c_2 n\) for some constant \(c_2\). (More details in class.)

Recurrence:

\[
T(1) = c_1 \\
T(n) \leq 2T(n/2) + c_2 n, \quad n \geq 2.
\]

Note: Recurrence identical to that for Merge Sort.

Solution: \(T(n) = O(n \log n)\).

A linear time algorithm:

Idea:

- Let \(b_i\) denote the sum of a max sum subsequence ending at \(x_i\) (\(1 \leq i \leq n\)).
• Required answer = \( \max\{b_1, b_2, \ldots, b_n\} \).

Note that \( b_1 = x_1 \). We can compute \( b_i \), \( 2 \leq i \leq n \), as follows.

\[
b_i = b_{i-1} + x_i \quad \text{if } b_{i-1} > 0 \\
= x_i \quad \text{otherwise.}
\]

**Pseudocode:**

• Array \( A \) contains the given sequence \( \langle x_1, x_2, \ldots, x_n \rangle \).

• Array \( B \) will store the values \( b_1 \) through \( b_n \).

2. for \( i = 2 \) to \( n \) do
   if \( (B[i-1] > 0) \)
   else
     \( B[i] = A[i] \)
3. Output the max value in array \( B \).

**Running time analysis:**

• Time for Step 1: \( O(1) \).

• Time for Step 2: \( O(n) \).

• Time for Step 3: \( O(n) \).

So, running time of algorithm = \( O(n) \).

**Recall:** \((c_1, c_2 : \text{constants})\)

For the recurrence

\[
T(1) = c_1 \\
T(n) \leq 2T(n/2) + c_2, \quad n \geq 2
\]

the solution is \( T(n) = O(n) \).

For the recurrence

\[
T(1) = c_1 \\
T(n) \leq 2T(n/2) + c_2 n, \quad n \geq 2
\]

the solution is \( T(n) = O(n \log n) \).

**Guidelines:**

• For an \( O(n) \) algorithm, the time for divide and combine steps should be \( O(1) \).

• For an \( O(n \log n) \) algorithm, the time for divide and combine steps should be \( O(n) \).
Multiplying long integers:

Input: Two \( n \)-bit integers \( X \) and \( Y \).
Output: The \( 2n \)-bit product \( XY \).

Traditional method:
- Computes \( n \) partial products (each with \( n \) bits)
- Time: \( O(n^2) \).

D-and-C Approach: (Assume \( n = 2^r \).)

\[
\begin{array}{c|c|c}
\hline
X & A & B \\
\hline
Y & C & D \\
\hline
\end{array}
\]

\[
X = 2^{n/2} A + B \quad Y = 2^{n/2} C + D
\]

\[
XY = 2^n AC + 2^{n/2} (AD + BC') + BD
\]

Computation involves
- (a) Four \( n/2 \)-bit multiplications.
- (b) Three \( 2n \)-bit additions.
- (c) Two \( O(n) \)-bit shifts.

\[
\begin{align*}
\text{Divide time} & : O(n). \\
\text{Combine (Steps (b) and (c))} & : O(n).
\end{align*}
\]

Recurrence:
\[
T(1) = c_1 \\
T(n) \leq 4T(n/2) + c_2 n, \quad n \geq 2.
\]

Solution: \( T(n) = O(n^2) \).

Improvement: Note that
\[
AD + BC = (A - B)(D - C) + AC + BD
\]

Therefore,
\[
XY = 2^n AC + [(A - B)(D - C) + AC + BD] 2^{n/2} + BD
\]

Now, the computation involves
- (a) Three \( n/2 \)-bit multiplications.
- (b) Six \( 2n \)-bit additions/subtractions.
- (c) Two \( O(n) \)-bit shifts.

Divide and combine time : \( O(n) \).
Recurrence:
\[ T(1) = c_1 \]
\[ T(n) \leq 3T(n/2) + c_3 n, \quad n \geq 2. \]

Solution: (To be presented in class)
\[ T(n) = O(n^{\log_2 3}). \]

Since \( \log_2 3 \approx 1.59 < 2 \), \( T(n) = o(n^2) \).

Logarithmic running time:

Observation: An algorithm has running time \( O(\log n) \) it takes \( O(1) \) time to cut the problem size by a fraction \( \alpha < 1 \).

Binary search: Cuts the problem size by the fraction \( 1/2 \) with \( O(1) \) work. Recurrence is:
\[ T(1) = c_1 \]
\[ T(n) \leq T(n/2) + c_2, \quad n \geq 2. \]

Solution: \( T(n) = O(\log n) \).

Computation of GCD:

Ref: Sections 31.1 and 31.2 of text.

GCD: Greatest Common Divisor.

Definition: Given two positive integers \( m \) and \( n \), \( \gcd(m, n) \) is the largest integer that divides both \( m \) and \( n \).

Examples:
- \( \gcd(32, 144) = 16 \).
- \( \gcd(15, 44) = 1 \). (15 and 44 are relatively prime.)

Lemma:
(a) For any positive integer \( n \), \( \gcd(n, 0) = n \).
(b) Let \( m \) and \( n \) be positive integers with \( m \geq n \). Then \( \gcd(m, n) = \gcd(n, m \mod n) \).

Proof idea for (b): Let \( S_1 \) denote the common divisors of \( m \) and \( n \). Let \( S_2 \) denote the common divisors of \( n \) and \( m \mod n \). It can be shown that \( S_1 = S_2 \).
Euclid’s Algorithm:

gcd (m, n) /* m > 0, n >= 0 */
1. while (n > 0)
   r = m mod n
   m = n;  n = r
2. return m

Example:

gcd(32, 144) = gcd(144, 32)
= gcd(32, 16)
= gcd(16, 0)
= 16.