2 Algorithm analysis method

Reading: MAW: Chapter 2

2.1 Asymptotic notation

- Used to compare growth rate or order of magnitude for increasing functions. “Asymptotic” deals with the behavior of functions in the limit, for sufficiently large values of variables.

- \( f(n) = O(g(n)) \) if \( \exists c, n_0 \) such that \( f(n) \leq cg(n) \) for \( n \geq n_0 \).

- \( f(n) = \Omega(g(n)) \) if \( \exists c, n_0 \) such that \( f(n) \geq cg(n) \) for \( n \geq n_0 \). (Or equivalently, \( g(n) = O(f(n)) \).)

- \( f(n) = \Theta(g(n)) \) if \( \exists c_1, c_2, n_0 \) such that \( c_2g(n) \leq f(n) \leq c_1g(n) \) for \( n \geq n_0 \). (Or equivalently, \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \).)

Remarks:
- Asymptotic notation: growth rate or order of magnitude, not exact value of the functions.

- \( O(\leq), \Omega(\geq), \) and \( \Theta(=) \).

- Ignores constant factors as well as lower-order terms, e.g., \( 10^9n^2 + 10^{10}n + 10^{100} = O(n^2) \).

- Ordered by increasing growth rate: constant \(<\) polylogarithmic \(<\) polynomial \(<\) exponential, i.e., 1, \( \log n \), \( \log^2 n \), \( n \), \( n\log n \), \( n^2 \), \( 2^n \).

- If \( T_1(n) = O(f(n)) \) and \( T_2(n) = O(g(n)) \), then \( T_1(n) + T_2(n) = \max\{O(f(n)), O(g(n))\} \) and \( T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n)) \).

Examples:
1. Compare \((\log n)^{100}\) and \(n^{0.01}\).
2. Order the following functions by increasing growth rate.
   \( \sqrt{n}, n^2, n!, \sqrt{n}\log n, 10n, 2^n, (\log n)^2, 2^{2^n}, \ln n, n^{\log\log n}, 1, n\log n, \sqrt{\log n}, n2^n \).

2.2 Time complexity of algorithms

- Factors that affect the actual running time of an algorithm: hardware (computer), software (language), people (programmer), and data (input). Can the analysis of algorithms ignore those complex factors?

- Measure the running time (time complexity) by the number of basic steps that the algorithm goes through.

- What is a basic step? \( +, -, \times, /, \) comparison, assignment, predicate evaluation, etc.. (**, while, and for are not.)

- Define the time complexity as a function of input size, which returns the number of basic steps. That is, time complexity \( T(n) \) is the number of basic steps for inputs of size \( n \).

- What is the input size? The amount of memory needed to store the input data. For example,

<table>
<thead>
<tr>
<th>Input</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td># of items</td>
</tr>
<tr>
<td>Matrix</td>
<td># of rows and columns</td>
</tr>
<tr>
<td>Graph</td>
<td># of vertices and edges</td>
</tr>
<tr>
<td>Integer</td>
<td># of bits</td>
</tr>
</tbody>
</table>

- What if there are millions of inputs with size \( n \)? We use the worst-case time complexity.

  - \( I_n \): Any input of size \( n \).
  - \( t(I_n) \): Time (number of basic steps) spent on \( I_n \) by the algorithm.
The worst-case time complexity of the algorithm is defined to be \( \max_{x \in A} \{ t(I_x) \} \). (But this is not how you calculate \( T(n) \).)

**How to analyze the worst-case time complexity of an algorithm:**

- Determine the input size \( n \);
- What is the worst case?
- Count the number of basic steps for that worst case and represent it as a function of \( n \). (Shortcuts may be taken.)
- Simplify the function by using the asymptotic notation.

**Example:** Compute \( \sum_{i=1}^n i^3 \).

```java
sum = 0
for i = 1 to n
    sum = sum + i * i * i
return sum
```

Time complexity: \( T(n) = 1 + (2n^2) + 4n + 1 = 6n + 4 = O(n) \).

Shortcut: \( T(n) = O(1) + O(n) = O(n) \).

**General rules:**

- Consecutive statements: These just add (meaning that the maximum is the one that counts). For example, \( O(1) + O(n^2) + O(n) = O(n^2) \).
- If \( C \) then \( S_1 \) else \( S_2 \): The running time is the running time of \( C \) plus the larger of the running times for \( S_1 \) and \( S_2 \).
- Loops/Nested loops: The running time is decided by the running time of the statement executed most (in the innermost loop), which can be computed by using summations. For example,
  ```java
  for i = 1 to n
      for j = 1 to n
          k ++
  ```

Time complexity: \( T(n) = \sum_{i=1}^n \sum_{j=1}^n 1 = \sum_{i=1}^n n = O(n^2) \).

### 2.3 An example: The maximum subsequence sum

- Given integers \( a_1, a_2, \ldots, a_n \), find the maximum value of \( \sum_{k=i}^j a_k \) for \( 1 \leq i \leq j \leq n \). (Also called the maximum subsequence sum.)

- For example, for input \(-2, 11, -4, 13, -5, -2\), the answer is 20 (\( i = 2 \) and \( j = 4 \)).

- **Algorithm 1:** Check all subsequences (blocks of consecutive items in the array).

  Idea: For each starting index \( i = 1, \ldots, n \), let the ending index \( j = i, \ldots, n \). Compute \( \sum_{k=i}^j a_k \) and keep track of the maximum sum.

```java
Input: a[1], ..., a[n]
Output: mss
mss = a[1]
for i = 1 to n
    for j = i to n
        sum = 0
        for k = i to j
            sum = sum + a[k]
        if sum > mss
            mss = sum
return mss
```
Time complexity: $O(n^3)$.

- Algorithm 2: Same as Algorithm 1, but uses a smart idea to compute $\sum_{k=i}^{j} a_k = \sum_{k=i}^{j-1} a_k + a_j$.

  Input: $a[1], \ldots, a[n]$  
  Output: $mss$

  \[
  mss = a[1] \\
  \text{for } i = 1 \text{ to } n \\
  \quad \text{sum} = 0 \\
  \quad \text{for } j = i \text{ to } n \\
  \quad \quad \text{sum} = \text{sum} + a[j] \\
  \quad \quad \text{if sum > mss} \\
  \quad \quad \quad mss = \text{sum} \\
  \text{return mss}
  \]

  Time complexity: $O(n^2)$.

- Algorithm 3: Divide-and-conquer.

  Idea: Divide the list $A$ into two equal-size sublists, $A_1$ and $A_2$. Determine the solutions, $mss_1$ and $mss_2$, for $A_1$ and $A_2$ recursively. Find the the maximum right subsequence sum, $mrss$, for $A_1$ and then the maximum left subsequence sum, $mlss$, for $A_2$. Finally, let $mss$ be the maximum of $mss_1$, $mss_2$, and $mrss + mlss$.

  Input: $a[1], \ldots, a[n]$  
  Output: $mss$

  Max_Subsequence_Sum(A)  
  if $|A| = 1$  
  \quad return $a[1]$  
  else  
  \quad $A \Rightarrow A_1$ and $A_2$, with $|A_1| = |A_2|$  
  \quad $mss_1 = \text{Max_Subsequence_Sum}(A_1)$  
  \quad $mss_2 = \text{Max_Subsequence_Sum}(A_2)$  
  \quad $mrss = \text{Max_RSubsequence_Sum}(A_1, |A_1|)$  
  \quad $mlss = \text{Max_LSubsequence_Sum}(A_2, |A_2|)$  
  \quad $mss = \max(mss_1, mss_2, mrss + mlss)$  
  \quad return $mss$

  Max_RSubsequence_Sum(B, m)  
  \[
  mrss = b[m] \\
  \text{sum} = 0 \\
  \text{for } i = m \text{ to } 1 \\
  \quad \text{sum} = \text{sum} + b[i] \\
  \quad \text{if sum > mrss} \\
  \quad \quad mrss = \text{sum} \\
  \text{return mrss}
  \]

  Max_LSubsequence_Sum(B, m)  
  \[
  mlss = b[1] \\
  \text{sum} = 0 \\
  \text{for } i = 1 \text{ to } m \\
  \quad \text{sum} = \text{sum} + b[i] \\
  \quad \text{if sum > mlss} \\
  \quad \quad mlss = \text{sum} \\
  \text{return mlss}
  \]
Time complexity:

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T\left(\frac{n}{2}\right) + n & \text{if } n \geq 2
\end{cases} \]

\[ T(n) = 2T\left(\frac{n}{2}\right) + n \\
= 2^2T\left(\frac{n}{4}\right) + 2n \\
= \ldots \\
= 2^kT\left(\frac{n}{2^k}\right) + kn \quad (n = 2^k) \\
= nT(1) + n\log n \\
= O(n\log n) \]

- Algorithm 4: Uses clever design.

Idea: Uses two pointers \textit{start} and \textit{end}. Let \textit{sum} be the corresponding sum of the block pointed by \textit{start} and \textit{end}.

- Case 1: \textit{start} = \textit{end} and \textit{sum} < 0.
  Set \textit{start} = \textit{end} + 1, \textit{end} = \textit{end} + 1, and \textit{sum} = 0.
- Case 2: \textit{start} = \textit{end} and \textit{sum} > 0.
  Set \textit{end} = \textit{end} + 1 and \textit{sum} = \textit{sum} + \textit{a}[\textit{end}].
- Case 3: \textit{start} < \textit{end} and \textit{sum} > 0
  Set \textit{end} = \textit{end} + 1 and \textit{sum} = \textit{sum} + \textit{a}[\textit{end}].
- Case 4: \textit{start} < \textit{end} and \textit{sum} < 0 (for the first time)
  \textit{a}[\textit{end}] must be a very negative number. In fact one can prove that it is so negative (small) that no subsequence with maximum sum includes it.
  Set \textit{start} = \textit{end} + 1, \textit{end} = \textit{end} + 1, and \textit{sum} = 0.

Input: \textit{a}[1], \ldots, \textit{a}[\textit{n}]
Output: \textit{mss}

\textit{mss} = \textit{a}[1]
\textit{sum} = 0
\textit{start} = 1
for \textit{end} = 1 to \textit{n}
    \textit{sum} = \textit{sum} + \textit{a}[\textit{end}]
    if \textit{sum} > \textit{mss}
        \textit{mss} = \textit{sum}
    if \textit{sum} < 0
        \textit{start} = \textit{end} + 1
        \textit{sum} = 0
return \textit{mss}

Time complexity: \(O(n)\).

- Compare \(O(n^3), O(n^2), O(n\log n)\), and \(O(n)\) by experiments (in seconds).

<table>
<thead>
<tr>
<th>(T(n))</th>
<th>(n = 10)</th>
<th>(n = 100)</th>
<th>(n = 1,000)</th>
<th>(n = 10,000)</th>
<th>(n = 100,000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n^3)</td>
<td>0.00103</td>
<td>0.47015</td>
<td>448.77</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>(n^2)</td>
<td>0.00045</td>
<td>0.01112</td>
<td>1.1233</td>
<td>111.13</td>
<td>NA</td>
</tr>
<tr>
<td>(n\log n)</td>
<td>0.00066</td>
<td>0.00486</td>
<td>0.05843</td>
<td>0.68631</td>
<td>8.0113</td>
</tr>
<tr>
<td>(n)</td>
<td>0.00034</td>
<td>0.00063</td>
<td>0.00333</td>
<td>0.03042</td>
<td>0.29832</td>
</tr>
</tbody>
</table>