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 $c_2 g(n) \leq f(n) \leq c_1 g(n)$ 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., $10^9 n^2 + 10^{10} n + 10^{100} = O(n^2)$.
  - 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,

Input	Size
List	# of items
Matrix	# of rows and columns
Graph	# of vertices and edges
Integer	# of bits

- 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.

- $T(n)$: The worst-case time complexity of the algorithm is defined to be $\max_{\forall I_n}\{t(I_n)\}$. (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 $\sum_{i=1}^{n} i^3$.

```
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,

```
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 $1 \le i \le j \le n$. (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.

```
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], ..., a[n]
Output: mss

mss = a[1]
for i = 1 to n
    sum = 0
    for j = i to n
        sum = sum + a[j]
        if sum > mss
           mss = sum
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], ..., a[n]
Output: mss

Max_Subsequence_Sum (A)
if |A| = 1
   return a[1]
else
   A => A1 and A2, with |A1| = |A2|
   mss1 = Max_Subsequence_Sum (A1)
   mss2 = Max_Subsequence_Sum (A2)
   mrss = Max_RSubsequence_Sum (A1,|A1|)
   mlss = Max_LSubsequence_Sum (A2,|A2|)
   mss = max{mss1, mss2, mrss+mlss}
   return mss

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

Max_LSubsequence_Sum(B, m)
mlss = b[1]
sum = 0
for i = 1 to m
    sum = sum + b[i]
    if sum > mlss
       mlss = sum
return mlss
```

Time complexity:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\frac{n}{2}) + n & \text{if } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n \\ &= 2^2 T(\frac{n}{2^2}) + 2n \\ &= \cdots\cdots \\ &= 2^k T(\frac{n}{2^k}) + kn \quad (n = 2^k) \\ &= nT(1) + n\log n \\ &= O(n\log n) \end{aligned}$$

- Algorithm 4: Uses clever design.

  Idea: Uses two pointers *start* and *end*. Let *sum* be the corresponding sum of the block pointed by *start* and *end*.

  - Case 1: $start = end$ and $sum < 0$.
    Set $start = end + 1$, $end = end + 1$, and $sum = 0$.
  - Case 2: $start = end$ and $sum > 0$.
    Set $end = end + 1$ and $sum = sum + a[end]$.
  - Case 3: $start < end$ and $sum > 0$
    Set $end = end + 1$ and $sum = sum + a[end]$.
  - Case 4: $start < end$ and $sum < 0$ (for the first time)
    $a[end]$ 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 $start = end + 1$, $end = end + 1$, and $sum = 0$.

```
Input:  a[1], ..., a[n]
Output: mss

mss = a[1]
sum = 0
start = 1
for end = 1 to n
    sum = sum + a[end]
    if sum > mss
       mss = sum
    if sum < 0
       start = end +1
       sum = 0
return mss
```

  Time complexity: $O(n)$.

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

$T(n)$	$n = 10$	$n = 100$	$n = 1{,}000$	$n = 10{,}000$	$n = 100{,}000$
n^3	0.00103	0.47015	448.77	NA	NA
n^2	0.00045	0.01112	1.1233	111.13	NA
$n\log n$	0.00066	0.00486	0.05843	0.68631	8.0113
n	0.00034	0.00063	0.00333	0.03042	0.29832