Complexity
&
Recurrences
Asymptotic Notations (Reminder)

- **$O$** - asymptotic upper bound
  \[
  O(g(n)) \equiv \{ f(n) \mid \exists c, n_0 > 0 : \forall n > n_0 \ 0 \leq f(n) \leq cg(n) \}
  \]

- **$\Omega$** - asymptotic lower bound
  \[
  \Omega(g(n)) \equiv \{ f(n) \mid \exists c, n_0 > 0 : \forall n > n_0 \ 0 \leq cg(n) \leq f(n) \}
  \]

- **$\Theta$** - asymptotic tight bound
  \[
  \Theta(g(n)) \equiv \{ f(n) \mid \exists c_1, c_2, n_0 > 0 : \forall n > n_0 \ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}
  \]

An analogy between comparison of functions and asymptotic notations:

\[
\begin{align*}
  f(n) &= \Theta(g(n)) \quad \approx \quad f(n) = g(n) \\
  f(n) &= O(g(n)) \quad \approx \quad f(n) \leq g(n) \\
  f(n) &= \Omega(g(n)) \quad \approx \quad f(n) \geq g(n)
\end{align*}
\]
תרגילים:

1. הפונקציות הבאות לכל האפשרות במשהיות סימוכן תכונה של \( n^3/1000 - 200n^2 + 1000n + 5 \):

\[ f_2(n) = O(g_2(n)) \quad \text{וכו} \quad f_1(n) = O(g_1(n)) \]

2. החכורה של הפריכי (ע"י דוגמא נגדי)

\[ f_1(n)f_2(n) = O(g_1(n)g_2(n)) \]

3. החכורה של הפריכי את התשענה המבואה:

\[ f(n_0) \leq g(n_0) \quad \text{אם ו-} \quad g(n) f(n) \quad \text{וכו} \quad n_0 \]

\[ f(n) = O(g(n)) \quad \text{אמנם} \quad n_0 \]

4. המה הגרוע ביותר המוכן חושר אמוריס "הopot וירדף של אולורימטה" \( \Omega(n^2) \) בגנוריב "הגרוע ישות?"
Example

• Find the complexity of the following code

```c
scanf("%d", &n);
k = 2;
for (i = 0; i < n; i++) {
    for (j = 0; j < i; j++) x += 1;
    while (k < i) k = k * k;
}
```

The outer `for` loop will be executed `n` times.
The inner `for` executed `i` times for each `i=0..n-1`:
\[
\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)
\]
The `while` loop will be executed `m` times:
\[
2^{2^m} \leq n \Rightarrow m = \log_2 \log_2 n = O(\log \log n)
\]

Total complexity: \( O(\log \log n) + O(n^2) = O(n^2) \)
Bubble Sort

As a recursive algorithm

\[
\text{Bubble-Sort}(A, i, n)
\]
\[
\text{if } i < n
\]
\[
\text{bubble smallest element to i-th place}
\]
\[
\text{Bubble-Sort}(A, i+1, n)
\]

To Bubble-Sort the array:  \(\text{Bubble-Sort}(A, 1, n)\)

\[
T(n) = (n-1) + T(n-1)
\]

\[
T(n) = (n-1) + T(n-1) = (n-1) + (n-2) + T(n-2) = \ldots = (n-1) + (n-2) + \ldots + 2 + 1
\]
Recurrences

- A Recurrence is an equation describing a function in terms of its value on smaller inputs:

\[
T(n) = \begin{cases} 
2T(\lfloor n/2 \rfloor) & \text{for } n > 1 \\
1 & \text{for } n = 1 
\end{cases}
\]

Our goal is to obtain an asymptotic bound on T, which is useful whenever T describes time/space complexity of some algorithm.
The Substitution Method

- We “guess” the solution and show the correctness by mathematical induction.

\[ T(n) = 2T\left(\lfloor n/2 \rfloor \right) + n \]

The solution we try is \( O(n \log n) \). We have to show, that there is a constant \( c \) such that

\[ T(m < n) \leq cm \log m \Rightarrow T(n) \leq cn \log n \]

\[ T(n) \leq c2\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor + n \]

\[ \leq cn \log n - cn \log 2 + n \]

\[ = cn \log n - (c \log 2 - 1)n \]

\[ \leq cn \log n, \quad \forall c > \frac{1}{\log 2} \]
The Iteration Method

- The idea is to expand the recurrence and express it as a summation of terms dependent only on $n$ and the initial conditions.

$$T(n) = n + 2T(n/3) = n + \frac{2}{3} n + 4T(n/9) = n + \frac{2}{3} n + \frac{4}{9} n + 8T(n/27) = ...$$

- We meet the boundary condition when $n/3^i \leq 1$, i.e. after $\lceil \log_3 n \rceil$ expansions

$$T(n) = n \sum_{i=0}^{\lceil \log_3 n \rceil} \left( \frac{2}{3} \right)^i + 2^{\log_3 n} \Theta(1) \leq n \sum_{i=0}^{\log_3 n + 1} \left( \frac{2}{3} \right)^i + 2^{\log_3 n} \Theta(1)$$

$$\leq n \sum_{i=0}^{\infty} \left( \frac{2}{3} \right)^i + n^{\log_3 2} \Theta(1) = 3n + O(n) = O(n)$$
Recursion Trees

• Consider: \( T(n) = T(n/3) + T(2n/3) + n \)

\[
\begin{align*}
T(n) & \quad n \\
T(n/3) & \quad T(2n/3) \\
\end{align*}
\]

Total: \( O(n \log n) \)
Example: Fibonacci Series

• Fibonacci series are defined recursively: \( a_{n+2} = a_{n+1} + a_n; \) for \( n \geq 2 \)
  
The starting values are: \( a_0 = 0 \)  
  \( a_1 = 1 \)

• Imagine a recursive function computing \( a_n \):
  
  ```python
  if n == 0 return 0;
  if n == 1 return 1;
  return f(n-1)+f(n-2);
  ```

• We shall build a recursion tree to show that this algorithm has exponential complexity.
• The boundary condition is first met by the rightmost leaf:

\[ T(n - 2d) < 2, \text{ when } d \text{ is the depth of that leaf} \text{ (= the height of the tree)}. \]

=> the height of the tree is at least \( n/2 \).

Since there are \( 2^i \) 1's in level \( i \), the total sum of operations performed is:

\[
T(n) \geq \sum_{i=0}^{n/2} 2^i = \frac{2^{n/2+1} - 1}{2 - 1} = \Omega(\sqrt{2}^n)
\]
• Efficient way: consider a function for computing \(a_n\),
which returns both \(a_n\) and \(a_{n-1}\) : \((a_n, a_{n-1}) \leftarrow f(n)\),
which is done with no additional cost (operations),
since you have to know \(a_{n-1}\), before you can compute \(a_n\).

```python
if n == 0 return (0,0);
if n == 1 return (1,0);
(An-1,An-2) <- f(n-1);
return (An-1 + An-2, An-1);
```

Note,
• each recursion iteration takes a constant amount of time (one addition),
• the boundary condition is met after n-1 iterations (we hit n=1).

Then, using the iteration method we see:

\[
T(n) = 1 + T(n-1) = 1 + 1 + T(n-2) = \ldots = \sum_{i=1}^{n-1} 1 + T(1) = O(n)
\]