4.1 Asymptotic Growth

- In the insertion-sort example we discussed that when analyzing algorithms we are
  - interested in worst-case running time as function of input size $n$
  - not interested in exact constants in bound
  - not interested in lower order terms

- A good reason for not caring about constants and lower order terms is that the RAM model is not
  completely realistic anyway (not all operations cost the same)

- We want to express rate of growth of standard functions:
  - the leading term with respect to $n$
  - ignoring constants in front of it

  \[
  \begin{align*}
  k_1 n + k_2 &\asymp n \\
  k_1 n \log n &\asymp n \log n \\
  k_1 n^2 + k_2 n + k_3 &\asymp n^2
  \end{align*}
  \]

- We also want to formalize e.g. that a $n \log n$ algorithms is better than a $n^2$ algorithm.

- $O$-notation (Big-$O$)
  - you have probably all seen it intuitively defined but we will now define it more carefully.

4.1.1 $O$-notation (Big-$O$)

\[
O(g(n)) = \{ f(n) : \exists c, n_0 > 0 \text{ such that } |f(n)| \leq c|g(n)| \forall n \geq n_0 \}
\]

- $O(\cdot)$ is used to asymptotically upper bound a function.
- $O(\cdot)$ is used to bound worst-case running time.
- Examples:
  - $1/3n^2 - 3n \in O(n^2)$ because $1/3n^2 - 3n \leq cn^2$ if $c \geq 1/3 - 3/n$ which holds for $c = 1/3$ and $n > 1$.
  - $k_1 n^2 + k_2 n + k_3 \in O(n^2)$ because $k_1 n^2 + k_2 n + k_3 < (k_1 + |k_2| + |k_3|)n^2$ and for $c > k_1 + |k_2| + |k_3|$ and $n \geq 1$, $k_1 n^2 + k_2 n + k_3 < cn^2$.
  - $k_1 n^2 + k_2 n + k_3 \in O(n^3)$ as $k_1 n^2 + k_2 n + k_3 < (k_1 + k_2 + k_3)n^3$ (Upper bound!).

- Note:
  - When we say “the running time is $O(n^2)$” we mean that the worst-case running time is $O(n^2)$ — best case might be better.
  - Use of $O$-notation often makes it much easier to analyze algorithms; we can easily prove the $O(n^2)$ insertion-sort time bound by saying that both loops run in $O(n)$ time.
  - We often abuse the notation a little:
    * We often write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$.
    * We often use $O(n)$ in equations: e.g. $2n^2 + 3n + 1 = 2n^2 + O(n)$ (meaning that $2n^2 + 3n + 1 = 2n^2 + f(n)$ where $f(n)$ is some function in $O(n)$).
    * We use $O(1)$ to denote constant time.

### 4.1.2 $\Omega$-notation (big-Omega)

$$\Omega(g(n)) = \{ f(n) : \exists c, n_0 > 0 \text{ such that } c|g(n)| \leq |f(n)| \forall n \geq n_0 \}$$

- $\Omega(\cdot)$ is used to asymptotically lower bound a function.
Examples:
- \( \frac{1}{3}n^2 - 3n = \Omega(n^2) \) because \( \frac{1}{3}n^2 - 3n \geq cn^2 \) if \( c \leq \frac{1}{3} - \frac{3}{n} \) which is true if \( c = 1/6 \) and \( n > 18 \).
- \( k_1 n^2 + k_2 n + k_3 = \Omega(n^2) \).
- \( k_1 n^2 + k_2 n + k_3 = \Omega(n) \) (lower bound!)

Note:
- When we say “the running time is \( \Omega(n^2) \)”, we mean that the best case running time is \( \Omega(n^2) \) — the worst case might be worse.

Insertion-sort:
- Best case: \( \Omega(n) \)
- Worst case: \( O(n^2) \)
- We can also say that the worst case running time is \( \Omega(n^2) \implies \) worst case running time is “precisely” \( n^2 \).

4.1.3 \( \Theta \)-notation (Big-Theta)

\[
\Theta(g(n)) = \{ f(n) : \exists c_1, c_2, n_0 > 0 \text{ such that } c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)| \forall n \geq n_0 \}
\]

- \( \Theta(\cdot) \) is used to asymptotically tight bound a function.
\[ f(n) = \Theta(g(n)) \text{ if and only if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \]

- Examples:
  - \( k_1 n^2 + k_2 n + k_3 = \Theta(n^2) \)
  - worst case running time of insertion-sort is \( \Theta(n^2) \)
  - \( 6n \log n + \sqrt{n} \log^2 n = \Theta(n \log n) \):
    * We need to find \( n_0, c_1, c_2 \) such that \( c_1 n \log n \leq 6n \log n + \sqrt{n} \log^2 n \leq c_2 n \log n \) for \( n > n_0 \)
    * \( c_1 n \log n \leq 6n \log n + \sqrt{n} \log^2 n \implies c_1 \leq 6 + \frac{\log n}{\sqrt{n}} \). If we choose \( c_1 = 6 \) and \( n_0 = 1 \).
    * \( 6n \log n + \sqrt{n} \log^2 n \leq c_2 n \log n \implies 6 + \frac{\log n}{\sqrt{n}} \leq c_2 \). Is it ok to choose \( c_2 = 7 \)? Yes, \( \log n \leq \sqrt{n} \) if \( n \geq 2 \).
      * So \( c_1 = 6, c_2 = 7 \) and \( n_0 = 2 \) works.

- Note:
  - We often think of \( f(n) = O(g(n)) \) as corresponding to \( f(n) \leq g(n) \).
  - Similarly, \( f(n) = \Theta(g(n)) \) corresponds to \( f(n) = g(n) \)
  - Similarly, \( f(n) = \Omega(g(n)) \) corresponds to \( f(n) \geq g(n) \)
  - One can also define \( o \) and \( \omega \)
    * \( f(n) = o(g(n)) \) corresponds to \( f(n) < g(n) \)
    * \( f(n) = \omega(g(n)) \) corresponds to \( f(n) > g(n) \)

4.1.4 Growth rate of standard functions

- Cormen book on Algorithms introduces standard functions in section 2.2 (we will introduce them as we need them):
  - Polynomial of degree \( d \): \( p(n) = \sum_{i=1}^{d} a_i \cdot n^i \) where \( a_1, a_2, \ldots, a_d \) are constants (and \( a_d > 0 \)). \( p(n) = \Theta(n^d) \)
  - “Growth order”: \( \log \log n, \log n, \sqrt{n}, n, n \log \log n, n \log n, n \log^2 n, n^2, n^3, 2^n \)
    - Growth rate of polynomials versus exponentials: \( \lim_{n \to \infty} \frac{n^k}{2^n} = 0. \)

Let us see the example of matrix multiplication. The straightforward pseudocode will be:
FOR $i = 1$ to $n$ DO
  FOR $j = 1$ to $n$ DO
    $C[i,j] = 0$
  FOR $k = 1$ to $n$
    $C[i,j] = C[i,j] + A[i,k] \cdot B[k,j]$
  OD
OD

This will take $O(n^3)$ time. But $C$ has $n^2$ entries, so we know that any algorithm that is used to multiply $A$ and $B$ will take at least $O(n^2)$ steps. Thus $O(n^2)$ is a lower bound. But can it be attained? We don't know. The best known algorithm has a time complexity of around $O(n^{2.4})$.

### 4.1.5 Abuse of Big-O

One must be careful while using the Big-O notation because even though theoretically the hidden constant is immaterial (asymptotically), it may be significant in practice.

For example, there is a simple parallel algorithm for sorting that runs in $O(\log^2 n)$ time using $O(n)$ processors. Because of the lower bound of $\Omega(n \log n)$ on sequential sorting, we know that using $O(n)$ processors we can't do better than $\Omega(\log n)$. So for a long time an open problem was whether one could sort $n$ numbers in $O(\log n)$ time using $(n)$ processors. Finally, Ajtai, Komlos and Szemeredi (1983) showed that $n$ numbers could be sorted in parallel and in $O(\log n)$ time. But the hidden constant in the running time was roughly $10^{10}$. For this algorithm to be better than the simpler one, we would need $\log n > 10^{10}$, so $n$ would have to be a gigantic $2^{10^{10}}$.

### 4.2 Recurrences

- As we will see later on with divide-and-conquer algorithms, the analysis of recursive algorithms leads to recurrence relations.
- Merge sort leads to the recurrence $T(n) = 2T(n/2) + \Theta(n)$
  - or rather, $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$
  - but for simplicity we will often simplify the formula by assuming $n$ to be of the form $2^k$ and ignore $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ operators.

### 4.3 Master Method

It be nice to have a general solution to $T(n) = aT(n/b) + f(n), \ T(1) = 1$. Master method has three cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(f(n))$.

There are some recurrences of the form given above that don't fit into any of the three cases and need special care. Try $T(n) = 2T(n/2) + n/\log n$. 
4.4 Divide and Conquer

Divide and Conquer is a design technique, based on the principle of dividing the original problem into smaller subproblems and solving them individually. The three main steps in any divide and conquer algorithm for a problem $P$ are:

1. **Divide**: $P$ into smaller problems $P_1, P_2, ..., P_k$.
2. **Conquer**: by solving the (smaller) subproblems recursively.
3. **Combine**: solutions to $P_1, P_2, ..., P_k$ into solutions for $P$.

4.4.1 Merge-sort

Using divide-and-conquer, we can obtain a merge-sort algorithm:

- **Divide**: Divide $n$ elements into two subsequences of $n/2$ elements each.
- **Conquer**: Sort the two subsequences recursively.
- **Combine**: Merge the two sorted subsequences.

Assume we have a procedure $\text{Merge}(A, p, q, r)$ which merges the sorted array $A[p..q]$ with the sorted array $A[q + 1..r]$ in $O(r - p)$ time. We can sort $A[1..n]$ as follows:

\begin{verbatim}
Merge-sort(A, p, r)
  If p < r then
    q = \lfloor (p + r)/2 \rfloor
    Merge-sort(A, p, q)
    Merge-sort(A, q + 1, r)
    Merge(A, p, q, r)
\end{verbatim}

![Figure 4.1: Running of Merge-sort](image)

**Analysis**

- To simplify things, let us assume that $n$ is a power of 2, i.e. $n = 2^k$ for some $k$. 
Running time of the procedure can be analyzed using a recurrence equation/relation.

\[ T(n) \leq 2T(n/2) + n + c \]

Note that this fits in case 2 of the Master method where \( a = b = 2 \) and \( f(n) = n + c \). Hence we get \( T(n) = \Theta(n \log n) \).

### 4.4.2 Binary Search

Another classic example of divide and conquer is binary search. This is exactly what people do when they look up someone’s telephone number in a telephone directory.

Suppose we have a list \( L \) and in this list we are searching for a value \( X \). If \( L \) has no special properties, if it is any old list, then there is no better way search it than to start at the beginning and go through the list one step at a time comparing each element to \( X \) in turn. This is called linear search - the time it takes (on average, and in the worst case) is linear, or \( O(n) \), in the length of the list.

But if \( L \) is a sorted list, there is a much faster way to search for \( X \):

1. Compare \( X \) to the middle value (\( M \)) in \( L \).
2. If \( X = M \) we are done.
3. If \( X > M \) we continue our search, but we can confine our search to the first half of \( L \) and entirely ignore the second half of \( L \).
4. If \( X < M \) we continue, but we confine ourselves to the second half of \( L \).

This is called binary search: during each iteration the length of the list we are looking in gets cut in half. Therefore we get the recurrence:

\[ T(n) = T(n/2) + 1 \]

If we use the Master method, we see \( a = 1, b = 2 \) and \( f(n) = 1 \) which fits in the third case and we get \( T(n) = O(\log n) \).

Note that the depth of the search tree reduces if we use a multiway search instead of a 2-way one as in a binary search tree. B-trees are an efficient data structures which support multiway searches.

### 4.5 Greedy Algorithms

Let us introduce a new design technique - Greedy method. Suppose we had to devise an algorithm to make change so as to minimize the number of coins. Assume the current system with pennies, nickels, dimes and quarters. Here a greedy strategy works very well. We keep on giving coins with the maximum denomination possible until we reach the desired amount. So to give change of 72 cents we would give 2 quarters, 2 dimes and 2 pennies. Is so happens that this strategy always works in this coin system. Now suppose there were no nickels in the coin system. So if we used this strategy to make change for 30 cents, we would end up giving one quarter and 5 pennies. But the best way is actually giving 3 dimes.

Greedy algorithms are powerful, but generally it is not very easy to prove their correctness. We will see more about Greedy algorithms in the next lecture.
4.6 Bibliographic notes

- Most of the material covered in this lecture can be found in the textbook *Introduction to Algorithms* by Cormen, T., Leiserson, C. and Stein, C.