Week 2

Divide and Conquer

1. Growth of Functions
2. Divide-and-Conquer
   - Min-Max-Problem
3. Tutorial
General remarks

- First we consider an important tool for the analysis of algorithms: **Big-Oh**.
- Then we introduce an important algorithmic paradigm: **Divide-and-Conquer**.
- We conclude by presenting and analysing two examples.

**Reading from CLRS for week 2**
- Chapter 2.3
- Chapter 3
Growth of Functions

- A way to describe behaviour of functions in the limit. We are studying asymptotic efficiency.
- Describe growth of functions.
- Focus on what’s important by abstracting away low-order terms and constant factors.
- How we indicate running times of algorithms.
- A way to compare “sizes” of functions:
  - \( \mathcal{O} \) corresponds to \( \leq \)
  - \( \Omega \) corresponds to \( \geq \)
  - \( \Theta \) corresponds to \( = \)

We consider only functions \( f, g : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \).
O-Notation

$O(g(n))$ is the set of all functions $f(n)$ for which there are positive constants $c$ and $n_0$ such that

$$f(n) \leq cg(n) \text{ for all } n \geq n_0.$$  

$g(n)$ is an asymptotic upper bound for $f(n)$.

If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$ (we will precisely explain this soon).
O-Notation Examples

\[ 2n^2 = O(n^3) \], with \( c = 1 \) and \( n_0 = 2 \).

Example of functions in \( O(n^2) \):

- \( n^2 \)
- \( n^2 + n \)
- \( n^2 + 1000n \)
- \( 1000n^2 + 1000n \)

Also

- \( n \)
- \( n/1000 \)
- \( n^{1.999999} \)
- \( n^2 / \log \log \log n \)
Ω-Notation

Ω(g(n)) is the set of all functions f(n) for which there are positive constants c and n₀ such that

\[ f(n) \geq cg(n) \quad \text{for all} \quad n \geq n₀. \]

\( g(n) \) is an asymptotic lower bound for \( f(n) \).
**Ω-Notation Examples**

\[ \sqrt{n} = \Omega(\lg n), \text{ with } c = 1 \text{ and } n_0 = 16. \]

Example of functions in \( \Omega(n^2) \):

- \( n^2 \)
- \( n^2 + n \)
- \( n^2 - n \)
- \( 1000n^2 + 1000n \)
- \( 1000n^2 - 1000n \)

Also

- \( n^3 \)
- \( n^{2.0000001} \)
- \( n^2 \lg \lg \lg n \)
- \( 2^{2^n} \)
\( \Theta(g(n)) \) is the set of all functions \( f(n) \) for which there are positive constants \( c_1, c_2 \) and \( n_0 \) such that

\[
c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \text{for all } n \geq n_0.
\]

\( g(n) \) is an asymptotic tight bound for \( f(n) \).
$\Theta$-Notation (cont’d)

**Examples 1**

\[ \frac{n^2}{2} - 2n = \Theta(n^2), \text{ with } c_1 = \frac{1}{4}, \ c_2 = \frac{1}{2}, \text{ and } n_0 = 8. \]

**Theorem 2**

\[ f(n) = \Theta(g(n)) \text{ if and only if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)). \]

Leading constants and lower order terms do not matter.
Asymptotic notation in equations

When on right-hand side

Θ(n²) stands for some anonymous function in the set Θ(n²).
2n² + 3n + 1 = 2n² + Θ(n) means 2n² + 3n + 1 = 2n² + f(n) for some f(n) ∈ Θ(n). In particular, f(n) = 3n + 1.

When on left-hand side

No matter how the anonymous functions are chosen on the left-hand side, there is a way to choose the anonymous functions on the right-hand side to make the equation valid.
Interpret 2n² + Θ(n) = Θ(n²) as meaning for all functions f(n) ∈ Θ(n), there exists a function g(n) ∈ Θ(n²) such that 2n² + f(n) = g(n).
Asymptotic notation chained together

\[ 2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2) \]

Interpretation:

- First equation: There exists \( f(n) \in \Theta(n) \) such that
  \[ 2n^2 + 3n + 1 = 2n^2 + f(n). \]
- Second equation: For all \( g(n) \in \Theta(n) \) (such as the \( f(n) \) used to make the first equation hold), there exists
  \( h(n) \in \Theta(n^2) \) such that
  \[ 2n^2 + g(n) = h(n). \]

Note

What has been said of “\( \Theta \)” on this and the previous slide also applies to “\( O \)” and “\( \Omega \)”. 
Example Analysis

**Insertion-Sort**\((A)\)

1. \textbf{for} \(j = 2\) \textbf{to} \(A\.\text{length}\)
2. \hspace{1em} \textit{key} = \(A[j]\)
3. \hspace{1em} \texttt{// Insert} \(A[j]\) \texttt{into sorted sequence} \(A[1..j-1]\).
4. \hspace{1em} \textit{i} = \(j-1\)
5. \hspace{2em} \textbf{while} \(i > 0\) \textbf{and} \(A[i] > \textit{key}\)
6. \hspace{3em} \(A[i+1] = A[i]\)
7. \hspace{3em} \textit{i} = \(i-1\)
8. \hspace{1em} \(A[i+1] = \textit{key}\)

The \textbf{for} -loop on line 1 is executed \(O(n)\) times; and each statement costs constant time, except for the \textbf{while} -loop on lines 5-7 which costs \(O(n)\).

Thus overall runtime is: \(O(n) \times O(n) = O(n^2)\).

\textbf{Note:} In fact, as seen last week, worst-case runtime is \(\Theta(n^2)\).
Divide-and-Conquer Approach

There are many ways to design algorithms.

For example, insertion sort is incremental: having sorted $A[1 \ldots j-1]$, place $A[j]$ correctly, so that $A[1 \ldots j]$ is sorted.

Divide-and-Conquer is another common approach:

**Divide** the problem into a number of subproblems that are smaller instances of the same problem.

**Conquer** the subproblems by solving them recursively.

  *Base case:* If the subproblem are small enough, just solve them by brute force.

**Combine** the subproblem solutions to give a solution to the original problem.
Naive Min-Max

Find minimum and maximum of a list $A$ of $n > 0$ numbers.

Naive-Min-Max($A$)

1. $least = A[1]$
2. for $i = 2$ to $A$.length
3. if $A[i] < least$
4. $least = A[i]$
5. $greatest = A[1]$
6. for $i = 2$ to $A$.length
7. if $A[i] > greatest$
8. $greatest = A[i]$
9. return ($least$, $greatest$)

The for-loop on line 2 makes $n - 1$ comparisons, as does the for-loop on line 6, making a total of $2n - 2$ comparisons.

Can we do better? Yes!
Divide-and-Conquer Min-Max

As we are dealing with subproblems, we state each subproblem as computing minimum and maximum of a subarray $A[p..q]$. Initially, $p = 1$ and $q = A\.length$, but these values change as we recurse through subproblems.

To compute minimum and maximum of $A[p..q]$:  

**Divide** by splitting into two subarrays $A[p..r]$ and $A[r+1..q]$, where $r$ is the halfway point of $A[p..q]$.

**Conquer** by recursively computing minimum and maximum of the two subarrays $A[p..r]$ and $A[r+1..q]$.

**Combine** by computing the overall minimum as the min of the two recursively computed minima, similar for the overall maximum.
Divide-and-Conquer Min-Max Algorithm

Initially called with $\text{MIN-MAX}(A, 1, A.\text{length})$.

$$\text{MIN-MAX}(A, p, q)$$

1. if $p = q$
2. return $(A[p], A[q])$
3. if $p = q−1$
5. return $(A[p], A[q])$
6. else return $(A[q], A[p])$
7. $r = \lfloor (p+q)/2 \rfloor$
8. $(\text{min1, max1}) = \text{MIN-MAX}(A, p, r)$
9. $(\text{min2, max2}) = \text{MIN-MAX}(A, r+1, q)$
10. return $(\text{min}(\text{min1, min2}), \text{max}(\text{max1, max2}))$

Note

- In line 7, $r$ computes the halfway point of $A[p \ldots q]$.
- $n = q − p + 1$ is the number of elements from which we compute the min and max.
Solving the Min-Max Recurrence

Let $T(n)$ be the number of comparisons made by $\text{MIN-MAX}(A, p, q)$, where $n = q - p + 1$ is the number of elements from which we compute the min and max.

Then $T(1) = 0$, $T(2) = 1$, and for $n > 2$:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 2.$$ 

Claim

$$T(n) = \frac{3}{2} n - 2 \quad \text{for} \quad n = 2^k \geq 2, \text{ i.e., powers of 2}.$$ 

Proof.

The proof is by induction on $k$ (using $n = 2^k$).

Base case: true for $k=1$, as $T(2^1) = 1 = \frac{3}{2} \cdot 2^1 - 2$.

Induction step: assuming $T(2^k) = \frac{3}{2} 2^k - 2$, we get

$$T(2^{k+1}) = 2T(2^k)+2 = 2 \left( \frac{3}{2} 2^k - 2 \right) + 2 = \frac{3}{2} 2^{k+1} - 2 \quad \square$$
Solving the Min-Max Recurrence (cont’d)

Some remarks:

1. If we replace line 7 of the algorithm by $r = p + 1$, then the resulting runtime $T'(n)$ satisfies $T'(n) = \lceil \frac{3n}{2} \rceil - 2$ for all $n > 0$.

2. For example, $T'(6) = 7$ whereas $T(6) = 8$.

3. It can be shown that at least $\lceil \frac{3n}{2} \rceil - 2$ comparisons are necessary in the worst case to find the maximum and minimum of $n$ numbers for any comparison-based algorithm: this is thus a lower bound on the problem.

4. Hence this (last) algorithm is provably optimal.
Big-Oh, Omega, Theta by examples

1. $5n + 111 = O(n)$ ? YES
2. $5n + 111 = O(n^2)$ ? YES
3. $5n + 111 = \Omega(n)$ ? YES
4. $5n + 111 = \Omega(n^2)$ ? NO
5. $5n + 111 = \Theta(n)$ ? YES
6. $5n + 111 = \Theta(n^2)$ ? NO
7. $2^n = O(3^n)$ ? YES
8. $2^n = \Omega(3^n)$ ? NO
9. $120n^2 + \sqrt{n} + 99n = O(n^2)$ ? YES
10. $120n^2 + \sqrt{n} + 99n = \Theta(n^2)$ ? YES
11. $\sin(n) = O(1)$ ? YES
Unfolding the recursion for Min-Max

We have

\[ T(n) = \begin{cases} 
0 & \text{if } n = 1 \\
1 & \text{if } n = 2 \\
T\left(\left\lceil \frac{n}{2} \right\rceil \right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + 2 & \text{else}
\end{cases} \]

1. \( T(1) = 0 \)
2. \( T(2) = 1 \)
3. \( T(3) = T(2) + T(1) + 2 = 1 + 0 + 2 = 3 \)
4. \( T(4) = T(2) + T(2) + 2 = 1 + 1 + 2 = 4 \)
5. \( T(5) = T(3) + T(2) + 2 = 3 + 1 + 2 = 6 \)
6. \( T(6) = T(3) + T(3) + 2 = 3 + 3 + 2 = 8 \)
7. \( T(7) = T(4) + T(3) + 2 = 4 + 3 + 2 = 9 \)
8. \( T(8) = T(4) + T(4) + 2 = 4 + 4 + 2 = 10 \)
9. \( T(9) = T(5) + T(4) + 2 = 6 + 4 + 2 = 12 \)
10. \( T(10) = T(5) + T(5) + 2 = 6 + 6 + 2 = 14. \)

We count 4 steps +1 and 5 steps +2 — we guess \( T(n) \approx \frac{3}{2} n \).
As you can see in the section on the min-max problem, for some input sizes we can validate the guess $T(n) \approx \frac{3}{2} n$.

One can now try to find a precise general formula for $T(n)$, however we see that we have $T(6) = 8$, while we can handle this case with 7 comparisons. So perhaps we can find a better algorithm?

And that is the case:

1. If $n$ is even, find the min-max for the first two elements using 1 comparison; if $n$ is odd, find the min-max for the first element using 0 comparisons.
2. Now iteratively find the min-max of the next two elements using 1 comparison, and compute the new current min-max using 2 further comparisons. And so on ....

This yields an algorithm using precisely $\lceil \frac{3}{2} n \rceil - 2$ comparisons. And this is precisely optimal for all $n$.

We learn: Here divide-and-conquer provided a good stepping stone to find a really good algorithm.