CZ 4016 - Advanced Topics in Algorithms

Suyash Lakhotia

AY 17/18, Semester 1

Contents

Chapter 1

Analysis

1.1 Growth of Functions

Assumption: All functions are non-negative.

1.1.1 O-Notation

$$O(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 \text{ s.t. } \forall n > n_0 : \\ 0 \leq f(n) \leq cg(n)\}$$

We say that $f(n)$ is **asymptotically upper bound** by $g(n)$, when $f(n) \in O(g(n))$ (also written as $f(n) = O(g(n))$).

1.1.2 Ω-Notation

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 \text{ s.t. } \forall n > n_0 : \\ 0 \leq cg(n) \leq f(n)\}$$

We say that $f(n)$ is **asymptotically lower bound** by $g(n)$, when $f(n) \in \Omega(g(n))$ (also written as $f(n) = \Omega(g(n))$).

1.1.3 Θ-Notation

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, \exists n_0 \text{ s.t. } \forall n > n_0 : \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

We say that $f(n)$ is **asymptotically tight bound** by $g(n)$, when $f(n) \in \Theta(g(n))$ (also written as $f(n) = \Theta(g(n))$).

$$f(n) = \Theta(g(n)) \implies f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

1.2 Alternative Notation

1.2.1 O-Notation

$$O(g(n)) = \left\{ f(n) \mid \limsup_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \sup_{m>n} \frac{f(m)}{g(m)} < \infty \right\}$$

1.2.2 Ω-Notation

$$\Omega(g(n)) = \left\{ f(n) \mid \liminf_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \inf_{m>n} \frac{f(m)}{g(m)} > 0 \right\}$$

1.2.3 Simplification

If $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, no need for sup and inf.

1.2.3.1 Corollary

If $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is well-defined and equals ξ, then:

- if $\xi < \infty$, then $f(n) = O(g(n))$
- if $\xi > 0$, then $f(n) = \Omega(g(n))$
- if $0 < \xi < \infty$, then $f(n) = \Theta(g(n))$

1.3 Comparing Functions

1.3.1 Transitivity

$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \implies f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \implies f(n) = \Omega(h(n))$$
$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n))$$

1.3.2 Reflexivity

$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$
$$f(n) = \Theta(f(n))$$

1.3.3 Symmetry

$$f(n) = \Theta(g(n)) \text{ iff } g(n) = \Theta(f(n))$$

1.3.4 Transpose Symmetry

$$f(n) = O(g(n)) \text{ iff } g(n) = \Omega(f(n))$$

1.4 Chaining & Composing Algorithms

1.4.1 Big-O

Let $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then:

$$f_1(n) + f_2(n) = O(g_1(n) + g_2(n)) = O(\max(g_1(n), g_2(n)))$$
$$f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

1.4.2 Big-Θ

Let $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then:

$$f_1(n) + f_2(n) = \Theta(g_1(n) + g_2(n)) = \Theta(\max(g_1(n), g_2(n)))$$
$$f_1(n) \cdot f_2(n) = \Theta(g_1(n) \cdot g_2(n))$$

1.5 Recurrence

1.5.1 Generic Substitution Method

Solve for $T(n) = T(n_1) + T(n_2) + T(n_3) + \ldots + T(n_k) + g(n)$, where $n_i < n$ and $g(n)$ is a known function.

Method: Guess a function, $T(n) = f(n)$ and substitute it into the recurrence step.

If $f(n)$ is too **high order**, then for large n, the l.h.s grows much faster than the r.h.s.

$$f(n) \gg f(n_1) + \ldots + f(n_k) + g(n)$$

If $f(n)$ is too **low order**, then for large n, the l.h.s grows much slower than the r.h.s.

$$f(n) \ll f(n_1) + \ldots + f(n_k) + g(n)$$

Correct $f(n)$ will **balance**:

$$f(n) \approx f(n_1) + \ldots + f(n_k) + g(n)$$

1.5.2 Iteration Method

Expand the recurrence steps and collapse to a form that incorporates the base step on the r.h.s. Then, simplify and conclude.

1.5.3 Master Method

Operates on any recurrence of the form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

- $a \geq 1$ and $b > 1$ are constants.
- $f(n)$ represents the cost of dividing the problem and/or combining the results of subproblems.
- $a \cdot T(\frac{n}{b})$ is the cost of solving a subproblem of size $\frac{n}{b}$ (floor or ceiling of this value i.e. an integer).

1.5.3.1 Theorem

- If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a})$, $T(n) = \Theta(n^{\log_b a} \lg n)$.
- If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$ and all $n > n_0$, $T(n) = \Theta(f(n))$.

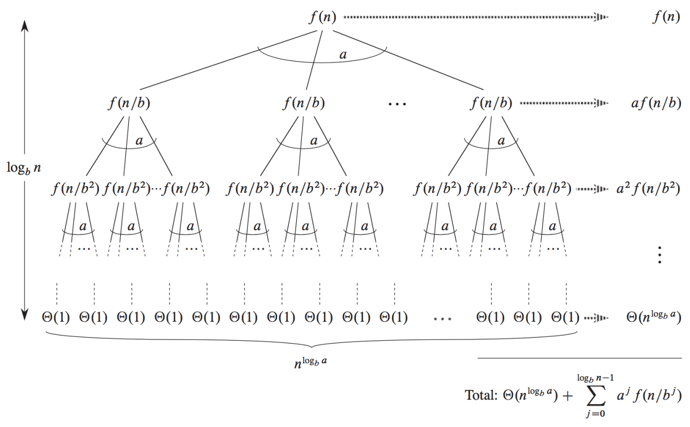

Figure 1.1: Master Theorem

1.6 Examples

See Lecture Notes for examples.

Chapter 2

Lower Bounds

The objective is to prove that any algorithm for a given problem will have to perform at least $g(n)$ operations.

- $f(n)$ is an **upper bound** to solve X, if there is some algorithm with runtime $\leq f(n)$.
- $g(n)$ is a **lower bound** to solve X, if any algorithm has runtime $\geq g(n)$.
- An algorithm is **optimal** if its upper bound matches the lower bound of the problem.
    - If $A(n) = O(f(n))$ and any other algorithm $A'(n) = \Omega(f(n))$, then nothing is faster than A.
    - Also means that $g(n)$ is a tight bound since no lower bound can be higher.

2.1 Why lower bounds?

- Stop looking for a better algorithm
- Structural problem analysis
    - Modeling assumption went awry
    - Generic features of hard instance
        - Focus speed-up efforts
        - Identify error tolerance
- Security
    - Cryptosystems
        - Breaking RSA requires efficient factoring of large numbers.
    - Social system manipulations
        - Skew voting results
        - Find triggers for cascade failures

2.2 Examples

See Lecture Notes for example derivations of lower bounds.

Chapter 3

Complexity Theory

- If a problem can be solved in poly-time, it's an **easy** problem.
- If a problem can be solved in exp-time, it's a **hard** problem.
- ***P* or Polynomial Time:** Class of problems that can be solved in poly-time on deterministic machines.
- ***NP* or Non-deterministic Polynomial Time:** Class of decision problems that can be solved in poly-time on non-deterministic machines.
    - Alternative Definition: Class of decision problems for which a solution can be verified by a poly-time algorithm on a deterministic machine.

3.1 Problem Types

- Decision Problems: Are there..? Is it possible to..?
- Counting Problems: How many..?
- Enumeration Problems: List all..that..
- Optimization Problems: What is the best..? What is the max/min..?

Any optimization problem can be cast as a decision problem given a witness solution (i.e. verification) or a quality bound.

3.2 Optimization to Decision Conversion

Optimization Problem P: Given a function f that maps $x, y \mapsto \boldsymbol{R}$. What is x so that $f(x, y)$ is minimal?

Decision Problem P_d: Given a function f that maps $x, y \mapsto \boldsymbol{R}$ and a quality bound d. Is there x such that $f(x, y) < d$?

If optimization problem P can be solved in $O(T)$, then the decision problem P_d can be solved in $O(T)$ as well since the optimal value is a special case.

If decision problem P_d can be solved in T time, then the optimization problem P can be solved in $O(T \lg N)$, where N is the range of possible values of P's objective function. Divide-and-conquer approach.

3.3 Reductions

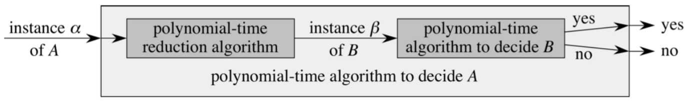
instance α of A
polynomial-time reduction algorithm
instance β of B
polynomial-time algorithm to decide B
yes
no
yes
no
polynomial-time algorithm to decide A

Figure 3.1: Reduction

A language L_1 is polynomial-time reducible to language L_2, denoted by $L_1 \leq_P L_2$ if there is a polynomial-time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ s.t. $\forall x \in \{0,1\}^*$ holds:

$$x \in L_1 \iff f(x) \in L_2$$

- The function f is the **reduction function**.
- The poly-time algorithm F to calculate f is a **reduction algorithm**.

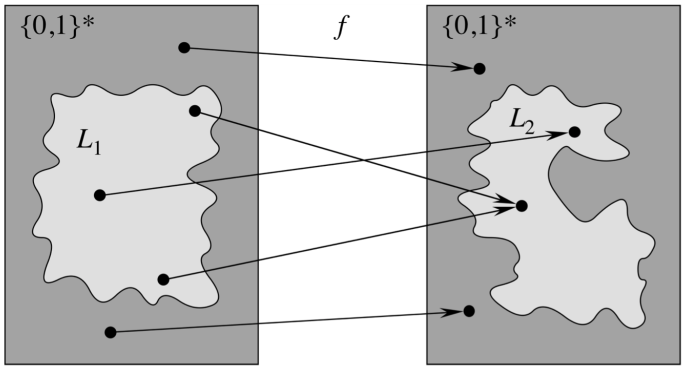

Figure 3.2: Reduction Mapping

3.3.1 Correctness

A reduction from A to B is correct if:

- Reduction runs in polynomial time.
- The answer to the A instance is "yes" iff it was reduced to "yes" instance of B.

3.3.2 NP-hard & NP-complete

- **NP-hard Problems:** A problem D is NP-hard if every problem in NP is poly-time reducible to D.
- **NP-complete Problems:** A problem D is NP-complete if it is in NP and NP-hard.

3.3.3 Completeness

A language L is **complete** for a language class C with respect to polynomial-time reductions if $L \in C$ and $L' \leq_P L$ for all $L' \in C$.

3.3.4 Proving NP-Completeness

To prove that $L \in NPC$:

- Prove that $L \in NP$
- Prove NP-hardness
  - Find a known NPC problem, Q
  - Show $Q \leq_P L$

3.4 Examples

See Lecture Notes for example derivations of reductions.

Chapter 4

Dynamic Programming

- Computation method based on sequencing, reuse and recombination of sub-problems.
- DP solution can support analysis of reducible problems.
- Provides first line of defense against complex problems.

4.1 Top-Down (Memoization)

```
F[0] = 0;
F[1] = 1;
F[2...n] = -1;

FibDP(n) {
    if (n == 0 || n == 1)
        return F[n];

    if (F[n] < 0)
        F[n] = FibDP(n - 1) + FibDP(n - 2);

    return F[n];
}
```

4.2 Bottom-Up

```
F[0] = 0;
F[1] = 1;
F[2...n] = -1;

FibDP(n) {
    for (i = 1 to i = n)
        F[i] = F[i - 1] + F[i - 2];

    return F[n];
}
```

4.3 DP Elements

- Optimal sub-structure
- Overlapping sub-problems
- Memoization (trading space for time)
- "Side information" for optimal solution reconstruction

A problem exhibits **optimal sub-structure**, if an optimal solution can be (re)constructed from optimal solutions to sub-problems.

4.4 DP Process

- Characterize the structure of the optimal solution
  - Solution definition by recursion (or iteration)
  - Identify any sharing in recursion (or iteration) steps
  - Optimal sub-structure + overlapping sub-problems
- Recursively/iteratively define the value of an optimal solution
- Compute the value of an optimal solution (exploit implicit DAG)
  - Memoization/Top-Down vs. Bottom-Up approach
- Construct an optimal solution from computed information
  - Extract the solution from the graph's topological order
  - Is there enough "side information" to recover argument of the optimum?

4.5 Examples

See Lecture Notes for example problems solved by dynamic programming.

Chapter 5

Search Techniques

Problem: Tree of possible sub-solutions to a problem may be huge, potentially infinite.

Solution: Build the tree as you search but stop building branches that are a priori bad.

5.1 Smarter Search

- S: State space of partial solutions
  - $s_0 \in S$: Initial (empty) solution
- $g : S \to \{0, 1\}$: Goal function
  - Is $s \in S$ a complete, legal, feasible solution?
- $g : S \to 2^S$: Expansion function
  - How can a partial solution be expanded?

5.2 Search (after construction)

```
Require: Set $G = (V, E)$,
  $V = \{s_0\}$, $E = \emptyset$
Require: Let $W = V$
  repeat
    Choose $w \in W$
    $W \leftarrow W \setminus \{w\}$
    $E \leftarrow E \cup \{(w, w') | w' \in f(w)\}$
    Merge $W$ and $f(w)$
  until $W == \emptyset$
  NOW SEARCH
```

Figure 5.1: Search (after construction)

Problem: Constructing the graph may go on indefinitely.

5.3 Search (during construction)

```
Require: Set $G = (V, E)$,
  $V = \{s_0\}$, $E = \emptyset$
Require: Let $W = V$
  repeat
    Choose $w \in W$
    if $g(w) == 1$ then
      return $w$
    $W \leftarrow W \setminus \{w\}$
    $E \leftarrow E \cup \{(w, w') | w' \in f(w)\}$
    Merge $W$ and $f(w)$
  until $W == \emptyset$
  Nothing found
```

Figure 5.2: Search (during construction)

5.4 Search (background construction)

```
Require: Let $W = \{s_0\}$
  repeat
    Choose $w \in W$
    if $g(w) == 1$ then
      return $w$
    $W \leftarrow W \setminus \{w\}$
    Merge $W$ and $f(w)$
  until $W == \emptyset$
  Nothing found
```

Figure 5.3: Search (background construction)

5.5 Even Smarter Search

Let W be an **ordered** set, a queue. Choosing $w \in W$ by removing the queue head:

- BFS Merge: $W = (W, f(W))$
- DFS Merge: $W = (f(W), W)$

5.6 Backtracking

But Merge and f can also **loose** some elements!

- Backtrack: Drop $w' \in f(w)$ if its subtree has no goal state.

5.6.1 Domain Formulation Requirement

Merge drops expansion from w if its subtree does not have the goal.

- $F_0(w) = \{w\}$, $F_{i+1}(w) = \bigcup_{s \in F_i(w)} f(s)$
- $F^*(w) = \bigcup_{i=0}^{\infty} F_i$
- If $g(s) == 0$ for all $s \in F^*(w)$, then Merge$(W, f(W)) = W$.

Alternatively,

- $F_0(w) = \{w\}$, $F_{i+1}(w) = \bigcup_{s \in F_i(w)} f(s)$
- $F^*(w) = \bigcup_{i=0}^{\infty} F_i$
- $H(s) == 0 \iff \forall s \in F^*(w) \;\; g(s) == 0$

where H is a heuristic function that can tell us if a goal does not exist in the extensions of a partial solution (i.e. the subtree of that partial solution). If $H(w) \;\; != 0$, we merge.

5.7 Branch-and-Bound

For a generic goal function, $g : S \to \mathbb{R} \;\bigcup\; \{\infty\}$ (i.e. optimization), two conditions need to be fulfilled:

1. Do not stop for any complete solution.
2. Instead of checking goal reachability, check a bound on its quality.

Assume that $g(w) = \infty$ means it is an infeasible solution.

Redefining H for minimization:

- $F_0(w) = \{w\}$, $F_{i+1}(w) = \bigcup_{s \in F_i(w)} f(s)$
- $F^*(w) = \bigcup_{i=0}^{\infty} F_i$
- $H(s) < \inf\{g(s) | s \in F^*(w)\}$

where H is the heuristic function that can determine the lower bound of all the solutions in the subtree to be expanded. The best possible solution thus far needs to be kept track of and we only merge if $H(w) < g(best)$.

Chapter 6

Computational Geometry

- A branch of computer science that studies algorithms for solving geometric problems.

Used for:

- Computer Graphics: hidden line removal, solid modeling
- Robotics: path planning, collision-avoidance
- VLSI Design: intersection detection
- Mobile Communications: mobile ad hoc routing, triangulation
- Geographic Information Systems, CAD/CAM, Statistics, Linguistic Analysis etc.

How to study it?

- Top-down w/ Memoization
  - Decompose a large problem and study individual components, until you understand the whole.
- Bottom-up
  - Study elementary (but initially boring) components and then keep combining them into infinitely more interesting problems.

6.1 Vectors, Segments, Planes & Turns

- **Point** $p = (x, y) \in \mathbb{R}^2$
- **Segment** between two points $\overline{p_1 p_2}$
- **Directed Segment** from p_0 to p_1: $\overrightarrow{p_0 p_1}$
  - A **vector** $\overrightarrow{p} = \overrightarrow{(x, y)}$ is a directed segment from point $(0, 0)$ to point $p = (x, y)$.
- A **convex combination** of points $p_1, \cdots, p_n$ is any point of the form:

$$p = \sum_{i=1}^{n} \alpha_i p_i, \text{ where } \alpha_i \geq 0 \text{ and } \sum_{i=1}^{n} \alpha_i = 1$$

- A **line** is given by an equation of the form $ax + by + c = 0$.
  - Positive half-plane are all points that satisfy $ax+by+c > 0$
  - Negative half-plane are all points that satisfy $ax+by+c < 0$
- **Cross Product** of vectors $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$:
  - If $p_1 \times p_2 > 0$, then p_1 is clockwise from p_2 w.r.t $(0, 0)$.
  - If $p_1 \times p_2 < 0$, then p_1 is counter-clockwise from p_2 w.r.t $(0, 0)$.
  - If $p_1 \times p_2 = 0$, then p_1 and p_2 are colinear.
  - The magnitude of the cross product can be interpreted as the positive area of the parallelogram contained by $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$.

$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1 \end{aligned}$$

6.2 Convex Hull

6.2.1 Convex Set

A set $C \subseteq \mathbb{R}^2$ is **convex** if for any $p_1, p_2 \in C$ holds $\overline{p_1 p_2} \subseteq C$.

6.2.2 Convex Hull

A **convex hull** of a set of points $S \subseteq \mathbb{R}^n$ is:

- the smallest convex set that includes S
- the intersection of all convex sets that include S
- the set of all convex combinations of points in S

6.2.3 Theorem 1 - Convex Polyhedra

For a finite S, the convex hull of S is a polyhedron and its vertices are a subset of S.

6.2.4 Corollary 2 - Caratheodory Theorem

Let S be a finite set in $\mathbb{R}^n$. For any $p \in CH(S)$, there is a subset $S' \subset S$ of size $|S'| = n + 1$ so that $p \in CH(S')$.

6.2.5 Lemma 3 - Compound Convex Hull

Given two finite sets $S_1, S_2 \in \mathbb{R}^n$, holds $CH(S_1 \cup S_2) = CH(CH(S_1) \cup CH(S_2))$.

Proof is given in the Lecture Notes.

6.3 Finding Convex Hulls

Refer to Lecture Notes for algorithms to find $CH(S)$ for finite $S \in \mathbb{R}^2$ and their respective proofs.

6.4 Path Planning

Refer to Lecture Notes.

6.5 Finding Closest Points

Refer to Lecture Notes.

Chapter 7

Math Formulae

7.1 Logarithms

$$\begin{aligned}
\log xy &= \log x + \log y \\
\log \frac{x}{y} &= \log x - \log y \\
\log^n x = (\log x)^n &= n \log x \\
\log_a a &= 1 \\
\log_a a^x &= x \\
a^{\log_a x} &= x \\
a^{(\log_b c)} &= c^{(\log_b a)} \\
\log_b a &= \frac{\log_c a}{\log_c b} \\
\log_b \left(\frac{1}{a}\right) &= -\log_b a \\
\log_b a &= \frac{1}{\log_a b}
\end{aligned}$$

7.2 Derivatives

$$\begin{aligned}
f(x) = c &\implies f'(x) = 0 \\
f(x) = x^n &\implies f'(x) = n \cdot x^{n-1} \\
f(x) = c \cdot g(x) &\implies f'(x) = c \cdot g'(x) \\
f(x) = g(x) \pm h(x) &\implies f'(x) = g'(x) \pm h'(x) \\
f(x) = g(x)h(x) &\implies f'(x) = g'(x)h(x) + g(x)h'(x) \\
f(x) = \frac{g(x)}{h(x)} &\implies f'(x) = \frac{g'(x)h(x) - g(x)h'(x)}{h(x)^2} \\
f(x) = g(h(x)) &\implies f'(x) = g'(h(x))h'(x)
\end{aligned}$$

7.2.1 Trig Derivatives

$$\begin{aligned}
f(x) = \sin(x) &\implies f'(x) = \cos(x) \\
f(x) = \cos(x) &\implies f'(x) = -\sin(x) \\
f(x) = \tan(x) &\implies f'(x) = \sec^2(x)
\end{aligned}$$

7.2.2 Exponential Derivatives

$$\begin{aligned}
f(x) = a^x &\implies f'(x) = \ln a \cdot a^x \\
f(x) = e^x &\implies f'(x) = e^x \\
f(x) = a^{g(x)} &\implies f'(x) = \ln a \cdot a^{g(x)} \cdot g'(x) \\
f(x) = e^{g(x)} &\implies f'(x) = e^{g(x)} \cdot g'(x) \\
f(x) = g(x)^{h(x)} &\implies h(x)g(x)^{h(x)-1}g'(x) + g(x)^{h(x)} \ln g(x)h'(x)
\end{aligned}$$

7.2.3 Logarithm Derivatives

$$\begin{aligned}
f(x) = \log_a x &\implies f'(x) = \frac{1}{x \cdot \ln a} \\
f(x) = \ln x &\implies f'(x) = \frac{1}{x} \\
f(x) = \log_a g(x) &\implies f'(x) = \frac{g'(x)}{g(x) \ln a} \\
f(x) = \ln g(x) &\implies f'(x) = \frac{g'(x)}{g(x)}
\end{aligned}$$

7.3 Arithmetic & Geometric Series

7.3.1 Sum of Arithmetic Series

$$S_n = \frac{n}{2}(a + l)$$

where a is the first term and l is the last (or n^{th}) term.

$$\therefore S_n = \frac{n}{2}(a + (n-1)d)$$

where d is the common difference.

7.3.2 Sum of Geometric Series

$$\begin{aligned}
S_n &= \frac{a(r^n - 1)}{r - 1} \qquad \text{if } r > 1 \\
S_n &= \frac{a(1 - r^n)}{1 - r} \qquad \text{if } r < 1
\end{aligned}$$

where a is the first term, r is the common ratio ($\neq 1$) and n is the number of terms.

7.3.2.1 Convergence

If $|r| < 1$, sum of an infinite series will be:

$$S_n = \frac{a}{1-r}$$

7.4 Limits

7.4.0.1 L'Hopital's Rule

If $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = 0, \pm\infty$, and $g'(n) \neq 0$, and also $\lim_{n\to\infty} \frac{f'(n)}{g'(n)}$ exists, then:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$