Algorithm Analysis

Correctness of Algorithms.
Efficiency of Algorithms
An algorithm is *correct* if, for any legal input, it halts (terminates) with the correct output.

A correct algorithm *solves* the given computational problem.

Automatic proof of correctness is not possible.

But there are practical techniques and rigorous formalisms that help to reason about the correctness of algorithms.
Partial and Total Correctness

- **Partial correctness**

  IF this point is reached, THEN this is the desired output

  Any legal input $\overset{\rightarrow}{\rightarrow}$ Algorithm $\overset{\rightarrow}{\rightarrow}$ Output

- **Total correctness**

  INDEED this point is reached, THEN this is the desired output

  Any legal input $\overset{\rightarrow}{\rightarrow}$ Algorithm $\overset{\rightarrow}{\rightarrow}$ Output
To prove partial correctness we associate a number of **assertions** (statements about the state of the execution) with specific checkpoints in the algorithm.

- E.g., $A[1], \ldots, A[k]$ form an increasing sequence

**Preconditions** – assertions that must be valid before the execution of an algorithm or a subroutine

**Postconditions** – assertions that must be valid after the execution of an algorithm or a subroutine
### Loop Invariants

- **Invariants** – assertions that are valid any time they are reached (many times during the execution of an algorithm, e.g., in loops)

- We must show three things about loop invariants:
  - **Initialization** – it is true prior to the first iteration
  - **Maintenance** – if it is true before an iteration, it remains true before the next iteration
  - **Termination** – when loop terminates the invariant gives a useful property to show the correctness of the algorithm
**Example of Loop Invariants (1)**

**Invariant:** at the start of each `for` loop, $A[1...j-1]$ consists of elements originally in $A[1...j-1]$ but in sorted order

```plaintext
for j=2 to length(A)
  do key=A[j]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
        A[i+1]:=key
```
**Example of Loop Invariants (2)**

**Invariant:** *at the start of each for loop, $A[1...j-1]$ consists of elements originally in $A[1...j-1]$ but in sorted order*

- **Initialization:** $j = 2$, the invariant trivially holds because $A[1]$ is a sorted array

```plaintext
for j=2 to length(A)
    do key=A[j]
        i=j-1
        while i>0 and A[i]>key
            do A[i+1]=A[i]
               i--
        A[i+1]:=key
```

--

DSA lecture 7 - T.U.Cluj-Napoca - M. Joldos
**Example of Loop Invariants (3)**

- **Invariant:** at the start of each for loop, \( A[1...j-1] \) consists of elements originally in \( A[1...j-1] \) but in sorted order

```
for j=2 to length(A)
  do key=A[j]
    i=j-1
    while i>0 and A[i]>key
      do A[i+1]=A[i]
      i--
    A[i+1]:=key
```

- **Maintenance:** the inner while loop moves elements \( A[j-1], A[j-2], \ldots, A[j-k] \) one position right without changing their order. Then the former \( A[j] \) element is inserted into \( k \)-th position so that \( A[k-1] \leq A[k] \leq A[k+1] \).

\[
A[1...j-1] \text{ sorted } + \ A[j] \rightarrow A[1...j] \text{ sorted}
\]
Example of Loop Invariants (3)

- **Invariant**: at the start of each for loop, $A[1...j-1]$ consists of elements originally in $A[1...j-1]$ but in sorted order

- **Termination**: the loop terminates, when $j = n+1$. Then the invariant states: "$A[1...n]$ consists of elements originally in $A[1...n]$ but in sorted order"

```plaintext
for j=2 to length(A)
    do key=A[j]
        i=j-1
        while i>0 and A[i]>key
            do A[i+1]=A[i]
                i--
        A[i+1]:=key
```
The running time of insertion sort is determined by a nested loop.

\[
\text{for } j \leftarrow 2 \text{ to } \text{length}(A) \\
\text{key} \leftarrow A[j] \\
i \leftarrow j - 1 \\
\text{while } i > 0 \text{ and } A[i] > \text{key} \\
\hspace{1em} A[i+1] \leftarrow A[i] \\
\hspace{1em} i \leftarrow i - 1 \\
\hspace{1em} A[i+1] \leftarrow \text{key}
\]

Nested loops correspond to summations

\[
\sum_{j=2}^{n} (j - 1) = O(n^2)
\]
Proof by Induction

- We want to show that property $P$ is true for all integers $n \geq n_0$
- **Basis**: prove that $P$ is true for $n_0$
- **Inductive step**: prove that if $P$ is true for all $k$ such that $n_0 \leq k \leq n - 1$ then $P$ is also true for $n$
- Example

\[
S(n) = \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \text{ for } n \geq 1
\]

- Basis

\[
S(1) = \sum_{i=0}^{1} i = \frac{1(1+1)}{2}
\]
Proof by Induction (2)

- Inductive Step

\[ S(k) = \sum_{i=1}^{k} i = \frac{k(k+1)}{2} \text{ for } 1 \leq k \leq n-1 \]

\[ S(n) = \sum_{i=0}^{n} i = \sum_{i=0}^{n-1} i + n = S(n-1) + n = \]

\[ = (n-1) \frac{(n-1+1)}{2} + n = \frac{(n^2 - n + 2n)}{2} = \]

\[ = \frac{n(n+1)}{2} \]
**Sum of odd numbers**

- **Problem:** Devise a recursive algorithm to add up the first $n$ odd numbers. That is, write a recursive algorithm that returns the following sum on input $n$

\[
\sum_{i=1}^{n}(2i - 1) = 1 + 3 + 5 + \ldots + 2n - 1.
\]

- **Solution**

```plaintext
SUM-ODD(n)
if n = 1 then
    return 1
else
    return [SUM-ODD(n - 1) + (2n - 1)]
```
Correctness of Sum of odd numbers

- **Claim:** $\text{SUM-ODD}(n)$ returns a value equal to $\sum_{i=1}^{n}(2i-1)$ for all natural numbers $n$.

- **Proof:** by induction on $n$.
  
  - **Base Case:** Let $n = 1$. When $\text{SUM-ODD}(n)$ is called with $n = 1$, the if condition is true, thus, $\text{SUM-ODD}(n)$ returns 1. Also,
    \[ \sum_{i=1}^{1}(2i-1) = 2 - 1 = 1. \]
  
  - **Inductive Hypothesis:** Assume that $\text{SUM-ODD}(k)$ returns a value
    \[ \sum_{i=1}^{k}(2i-1). \]
  
  - **Inductive Conclusion** (to show): Assume that $\text{SUM-ODD}(k+1)$ returns a value equal to
    \[ \sum_{i=1}^{k+1}(2i-1). \]
Correctness of Sum of odd numbers

- Inductive Step: First note that $k > 1$. Thus, the else case is executed.

- Treating a *return* statement as an assignment we have:

\[
\text{SUM-ODD}(k + 1) = \text{SUM-ODD}((k + 1) - 1) + 2(k + 1) - 1
\]

\[= \text{SUM-ODD}(k) + 2k + 1\]

\[= \sum_{i=1}^{k} (2i - 1) + 2k + 1\]

\[= \sum_{i=1}^{k+1} (2i - 1)\]
**Problem:** Determine whether a number $x$ is present in a *sorted* array $A[a..b]$

**Binary Search Solution:**

- Compare the middle element $mid$ to $x$
- If $x = mid$, stop
- If $x < mid$, throw away larger elements
- If $x > mid$, throw away smaller elements
- If there is no element left, $x$ is not in the array
**Binary Search Code**

BinarySearch(A, a, b, x)

1. If $a > b$ then
2. return false
3. else
4. \( \text{mid} \leftarrow \lfloor (a+b)/2 \rfloor \)
5. If $x = A[mid]$ then
6. return true
7. If $x < A[mid]$ then
8. return BinarySearch(A, a, mid−1, x)
9. else
10. return BinarySearch(A, mid+1, b, x)

Running time calculations:
On each iteration, more than half of elements are removed.

Program will run while
\[ n^{(0.5)} k > 1 \]
\[ k < \lg n \]
Correctness of Binary Search

- How do you know if it BinarySearch works correctly?

- First we need to precisely state what the algorithm does through the precondition and postcondition

  - The precondition states what may be assumed to be true initially:
    - Pre: \( a \leq b + 1 \) and \( A[a..b] \) is a sorted array
    - \( \text{found} = \text{BinarySearch}(A, a, b, x) \);

  - The postcondition states what is to be true about the result
    - Post: \( \text{found} = x \in A[a..b] \) and \( A \) is unchanged
Correctness of Recursive Algorithms

- Proof must take us from the precondition to the postcondition.
  - **Base case:** \( n = b - a + 1 = 0 \)
    - The array is empty, so \( a = b + 1 \)
    - The test \( a > b \) succeeds and the algorithm correctly returns false
  - **Inductive step:** \( n = b - a + 1 > 0 \)
    - **Inductive hypothesis:**
      Assume BinarySearch(\( A, a', b', x \)) returns the correct value for all \( j \) such that \( 0 \leq j \leq n - 1 \) where \( j = b' - a' + 1 \).
Correctness of Recursive Algorithms

• The algorithm first calculates $mid = \lfloor (a + b) / 2 \rfloor$, thus $a \leq mid \leq b$.

• If $x = A[mid]$, clearly $x \in A[a..b]$ and the algorithm correctly returns true.

• If $x < A[mid]$, since $A$ is sorted (by the precondition), $x$ is in $A[a..b]$ if and only if it is in $A[a..mid - 1]$. By the inductive hypothesis, BinarySearch$(A, a, mid - 1, x)$ will return the correct value since $0 \leq (mid - 1) - a + 1 \leq n - 1$.

• The case $x > A[mid]$ is similar

○ We have shown that the postcondition holds if the precondition holds and BinarySearch is called.
Summing an Array

- Problem: Given an array of numbers $A[a..b]$ of size $n = b - a + 1 \geq 0$, compute their sum.

// Pre: $a \leq b + 1$
1 $i \leftarrow a$, $sum \leftarrow 0$
2 while $i \neq b + 1$ do  // exit condition, called guard $G$
3 $sum \leftarrow sum + A[i]$
4 $i \leftarrow i + 1$
// Post: $sum = \sum_{j=a}^{b} A[j]$
Correctness of Iterative Algorithms

- The key step in the proof is the invention of a condition called the **loop invariant**, which is supposed to be true at the beginning of an iteration and remains true at the beginning of the next iteration.
- The steps required to prove the correctness of an iterative algorithm is as follows:
  1. Guess a condition \( I \)
  2. Prove by induction that \( I \) is a loop invariant
  3. Prove that \( I \land \neg G \Rightarrow \text{Postcondition} \)
  4. Prove that the loop is guaranteed to terminate
Correctness of Iterative Algorithms

- In the example, we know that when the algorithm terminates with \( i = b + 1 \), the following condition must hold:
  \[
  sum = \sum_{j=a}^{i-1} A[j]
  \]

- Use as invariant. Show that at the beginning of the the \( k \)-th loop, the condition holds

- **Base Case**: \( k = 1 \)
  - Initialized to \( i = a \) and \( sum = 0 \). Therefore
  \[
  \sum_{j=a}^{i-1} A[j] = 0
  \]

- **Inductive hypothesis**: Assume at the start of the loop’s \( k \)-th execution
  \[
  sum = \sum_{j=a}^{i-1} A[j]
  \]
Correctness of Iterative Algorithms

- Let \( \text{sum}' \) and \( i' \) be the values of the variables \( \text{sum} \) and \( i \) at the beginning of the \((k+1)\)-st iteration.

- In the \( k \)-th iteration, the variables were changed as follows:
  - \( \text{sum}' = \text{sum} + A[i] \)
  - \( i' = i + 1 \)

- Using the inductive hypothesis, we have
  \[
  \text{sum}' = \text{sum} + A[i] = \sum_{j=a}^{i-1} A[j] + A[i] = \sum_{j=a}^{i} A[j] = \sum_{j=a}^{i'-1} A[j]
  \]
Correctness of Iterative Algorithms

- We have proven the loop invariant $I$.
- Now we must show: $I \land \neg G \Rightarrow Postcondition$
  - We have $\neg G \Rightarrow i = b + 1$. Substituting into the invariant:
    $$\text{sum} = \sum_{j=a}^{b+1-1} A[j] = \sum_{j=a}^{b} A[j] \equiv Postcondition$$
- Remains to show that $G$ will eventually be false.
  - Note that $i$ is monotonically increasing since it is incremented inside the loop and not modified elsewhere.
  - From the precondition, $i$ is initialized to $a \leq b + 1$. 
Summary on Correctness

- How to prove correctness of *recursive algorithm*:
  - Induction

- Proving an algorithm:
  - Precondition
  - Postcondition

- How to prove correctness of *iterative algorithm*:
  - Identify a loop invariant condition, and prove it
  - Show that the invariant and terminating condition implies the postcondition
  - Show that the loop is guaranteed to terminate.
Efficiency of algorithms

- Algorithms for solving the same problem can differ dramatically in their efficiency.
- Much more significant than the differences due to hardware and software.

Comparison of two sorting algorithms ($n=10^6$ numbers):

- Insertion sort: $c_1 n^2$
- Merge sort: $c_2 n \ (\lg n)$
- Best programmer ($c_1=2$), machine language, one billion/second computer.
- Bad programmer ($c_2=50$), high-language, ten million/second computer.
- $2 (10^6)^2$ instructions/$10^9$ instructions per second $= 2000$ seconds.
- $50 (10^6 \lg 10^6)$ instructions/$10^7$ instructions per second $\approx 100$ seconds.
- Thus, merge sort on B is 20 times faster than insertion sort on A!
- If sorting ten million number, 2.3 days VS. 20 minutes.
Asymptotic Efficiency of Recurrences

- Find the asymptotic bounds of recursive equations.
  - Substitution method
  - Recursive tree method
  - Master method (master theorem)
    - Provides bounds for: \( T(n) = aT(n/b) + f(n) \) where
      - \( a \geq 1 \) (the number of subproblems).
      - \( b > 1 \), \( n/b \) is the size of each subproblem).
      - \( f(n) \) is a given function.
The Substitution Method

- Two steps:
  - Guess the form of the solution.
    - By experience, and creativity.
    - By some heuristics.
      - If a recurrence is similar to one you have seen before.
        - $T(n)=2T(\lfloor n/2 \rfloor)+17+n$, similar to $T(n)=2T(\lfloor n/2 \rfloor)+n$, guess $O(n \lg n)$.
  - Prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty.
    - For $T(n)=2T(\lfloor n/2 \rfloor)+n$, prove lower bound $T(n)=\Omega(n)$, and prove upper bound $T(n)=O(n^2)$, then guess the tight bound is $T(n)=O(n \lg n)$.
  - By recursion tree.
  - Use mathematical induction to find the constants and show that the solution works.
Substitution Method Example

- Solve $T(n) = 2T(\lfloor n/2 \rfloor) + n$
- Guess the solution: $T(n) = O(n \lg n)$,
  - i.e., $T(n) \leq cn \lg n$ for some $c$.
- Prove the solution by induction:
  - Suppose this bound holds for $\lfloor n/2 \rfloor$, i.e.,
    - $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)$.
  - $T(n) \leq 2(c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)) + n$
    - $\leq cn \lg (n/2) + n$
    - $= cn \lg n - cn \lg 2 + n$
    - $= cn \lg n - cn + n$
    - $\leq cn \lg n$ (as long as $c \geq 1$)
Substitution Method Example

- Boundary (base) Condition
  - In fact, \( T(n) = 1 \) if \( n=1 \), i.e., \( T(1)=1 \).
  - However, \( cn \lg n = c \times 1 \times \lg 1 = 0 \), which is odd with \( T(1)=1 \).

- Take advantage of asymptotic notation: it is required \( T(n) \leq cn \lg n \) hold for \( n \geq n_0 \), where \( n_0 \) is a constant of our choosing.

- Select \( n_0 = 2 \), thus, \( n = 2 \) and \( n = 3 \) as our induction bases. It turns out any \( c \geq 2 \) suffices for base cases of \( n = 2 \) and \( n = 3 \) to hold.
Guess is correct, but induction proof does not work.
Problem is that inductive assumption not strong enough.
Solution: revise the guess by subtracting a lower-order term.
Example: \( T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \).

- Guess \( T(n) = O(n) \), i.e., \( T(n) \leq cn \) for some \( c \).
- However, \( T(n) \leq c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1 = cn + 1 \), which does not imply \( T(n) \leq cn \) for any \( c \).
- Attempting \( T(n) = O(n^2) \) will work, but overkill.
- New guess \( T(n) \leq cn - b \) will work as long as \( b \geq 1 \).
Avoid pitfalls

- It is easy to guess $T(n) = O(n)$ (i.e., $T(n) \leq cn$) for $T(n) = 2T(\lfloor n/2 \rfloor) + n$.
- And wrongly prove:
  - $T(n) \leq 2(c \lfloor n/2 \rfloor) + n$
    - $\leq cn + n$
    - $= O(n)$. ❯ wrongly !!!!
- Problem is that it does not prove the exact form of $T(n) \leq cn$. 
Suppose $T(n) = 2T(\sqrt{n}) + \log n$.

Rename $m = \log n$. So $T(2^m) = 2T(2^{m/2}) + m$.

Rename $S(m) = T(2^m)$, so $S(m) = 2S(m/2) + m$.

Which is similar to $T(n) = 2T(\lfloor n/2 \rfloor) + n$.

So the solution is $S(m) = O(m \log m)$.

Changing back to $T(n)$ from $S(m)$, the solution is $T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n)$.
The Recursion–tree Method (I)

Steps:

1. Draw the tree based on the recurrence

2. From the tree determine:
   • # of levels in the tree
   • cost per level
   • # of nodes in the last level
   • cost of the last level (which is based on the number found in 2c)

3. Write down the summation using $\Sigma$ notation – this summation sums up the cost of all the levels in the recursion tree
The Recursion–tree Method (II)

Steps:

4. Recognize the sum or look for a closed form solution for the summation created in 3).

5. Apply that closed form solution to your summation coming up with your “guess” in terms of Big-O, or Θ, or Ω (depending on which type of asymptotic bound is being sought).

6. Then use Substitution Method or Master Method to prove that the bound is correct.
The Recursion–tree Method (III)

- **Idea:**
  - Each node represents the cost of a single subproblem.
  - Sum up the costs with each level to get level cost.
  - Sum up all the level costs to get total cost.

- Particularly suitable for divide-and-conquer recurrence.

- Best used to generate a good guess, tolerating “sloppiness”.

- If trying to compute cost as exact as possible, then used as direct proof.
Recursion Tree for $T(n) = 3T\left\lfloor \frac{n}{4} \right\rfloor + \Theta(n^2)$

(a) $T(n) = cn^2$
(b) $T(n/4)T(n/4)T(n/4)$
(c) $c(n/4)^2$
(d) $\log_4 n$

Total $O(n^2)$
Solution to $T(n)=3T\left(\lfloor n/4 \rfloor\right)+\Theta(n^2)$

- The height is $\log_4 n$.
- $\#\text{leaf nodes} = 3^{\log_4 n} = n^{\log_4 3}$. Leaf node cost: $T(1)$.
- Total cost $T(n)=cn^2+(3/16)\ cn^2+(3/16)^2\ cn^2+$
  $\ldots+(3/16)^{\log_4 (n-1)}\ cn^2+ \Theta(n^{\log_4 3})$
  $=(1+3/16+(3/16)^2+\ldots+(3/16)^{\log_4 n-1})\ cn^2 + \Theta(n^{\log_4 3})$
  $<(1+3/16+(3/16)^2+\ldots+(3/16)^m+\ldots)\ cn^2 + \Theta(n^{\log_4 3})$
  $=(1/(1-3/16))\ cn^2 + \Theta(n^{\log_4 3})$
  $=16/13cn^2 + \Theta(n^{\log_4 3})$
  $=O(n^2)$. 
Prove the previous Guess

- \( T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2) = O(n^2). \)
- Show \( T(n) \leq dn^2 \) for some \( d \).
- \( T(n) \leq 3(d \lfloor n/4 \rfloor^2) + cn^2 \)
  \[ \leq 3(d (n/4)^2) + cn^2 \]
  \[ = 3/16(dn^2) + cn^2 \]
  \[ \leq dn^2, \text{ as long as } d \geq (16/13)c. \]
Master Method/Theorem

- Used for recurrences of the form
  \[ T(n) = aT(n/b)+f(n), \ n/b \text{ may be } \lfloor n/b \rfloor \text{ or } \lceil n/b \rceil. \]
  where \( a \geq 1, \ b > 1, \ f(n) \text{ be a function.} \)

- Three cases:
  1. If \( f(n)=O(n^{\log_b a-\varepsilon}) \) for some \( \varepsilon > 0 \),
     then \( T(n)=\Theta(n^{\log_b a}). \)
  2. If \( f(n)=\Theta(n^{\log_b a}), \) then \( T(n)=\Theta(n^{\log_b a \lg n}). \)
  3. If \( f(n)=\Omega(n^{\log_b a+\varepsilon}) \) for some \( \varepsilon > 0, \) and
     if \( af(n/b) \leq cf(n) \) for some \( c < 1 \) and all
     sufficiently large \( n, \) then \( T(n)=\Theta(f(n)). \)
**Implications of Master Theorem**

- Comparison between $f(n)$ and $n^{\log_b a}$ ($<, =, >$)
- Must be asymptotically smaller (or larger) by a polynomial, i.e., $n^\varepsilon$ for some $\varepsilon > 0$.
- In case 3, the “regularity” must be satisfied, i.e., $af(n/b) \leq cf(n)$ for some $c < 1$.
- There are gaps
  - between 1 and 2: $f(n)$ is smaller than $n^{\log_b a}$, but not polynomially smaller.
  - between 2 and 3: $f(n)$ is larger than $n^{\log_b a}$, but not polynomially larger.
  - in case 3, if the “regularity” fails to hold.
Application of Master Theorem

- $T(n) = 9T(n/3)+n$;
  - $a = 9$, $b = 3$, $f(n) = n$
  - $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$
  - $f(n) = O(n^{\log_3 9-\epsilon})$ for $\epsilon = 1$
  - By case 1, $T(n) = \Theta(n^2)$.

- $T(n) = T(2n/3)+1$
  - $a = 1$, $b = 3/2$, $f(n) = 1$
  - $n^{\log_b a} = n^{\log_{3/2} 1} = \Theta(n^0) = \Theta(1)$
  - By case 2, $T(n) = \Theta(\log n)$. 

Application of Master Theorem

- \( T(n) = 3T(n/4)+n\lg n \);
  - \( a=3, b=4, f(n) = n\lg n \)
  - \( n^{\log_b a} = n^{\log_4 3} = \Theta(n^{0.793}) \)
  - \( f(n) = \Omega(n^{\log_4 3+\varepsilon}) \) for \( \varepsilon \approx 0.2 \)
  - Moreover, for large \( n \), the “regularity” holds for \( c = 3/4 \).
    - \( af(n/b) = 3(n/4)\lg (n/4) \leq (3/4)n\lg n = cf(n) \)
    - By case 3, \( T(n) = \Theta(f(n)) = \Theta(n\lg n) \).
Exception to Master Theorem

- \( T(n) = 2T(n/2) + n \log n; \)
  - \( a = 2, \ b = 2, \ f(n) = n \log n \)
  - \( n^{\log_b a} = n^{\log_2 2} = \Theta(n) \)
  - \( f(n) \) is asymptotically larger than \( n^{\log_b a} \), but not polynomially larger because
  - \( f(n)/n^{\log_b a} = \log n \), which is asymptotically less than \( n^\varepsilon \) for any \( \varepsilon > 0 \).
  - Therefore, this is a gap between 2 and 3.
Gaps

\[ f(n), \text{ case 3, at least polynomially larger} \]

\[ n^{\log_b a} \]

\[ f(n), \text{ case 2: within constant distances} \]

\[ \text{Gap between case 3 and 2} \]

\[ \text{Gap between case 1 and 2} \]

\[ f(n), \text{ case 1, at least polynomially smaller} \]

Notes: 1. for case 3, the regularity also must hold.

2. if \( f(n) \), is \( \log n \) smaller, then fall in gap in 1 and 2

3. if \( f(n) \) is \( \log n \) larger, then fall in gap in 3 and 2

4. if \( f(n) = \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \).
Reading

- AHU, chapter 8
- Preiss, chapter: Algorithm Analysis, Asymptotic Notation
- CLR, CLRS, chapters 2, 3, 4
- Notes