Analysis of Algorithms

Analysis of Algorithms: Issues

- Correctness/Optimality
- Running time ("time complexity")
- Memory requirements ("space complexity")
- Power
- I/O utilization
- Ease of implementation
- ...
Worst Case Time-Complexity

• **Definition:** The worst case time-complexity of an algorithm $A$ is the asymptotic running time of $A$ as a function of the size of the input, when the input is the one that makes the algorithm slower in the limit.

• How do we **measure** the running time of an algorithm?

Python (the language)

• We will use **python code** to describe algorithms (sometime mixed w English)

• Python is
  – High-level (easy to read/use/learn)
  – Object-oriented
  – Interpreted (but can be compiled)
  – Portable
  – Free/open-source
Python: an example

- Algorithm for finding the maximum element of an array

```python
def iMax(A):
    currentMax = A[0]
    for i in range(1, len(A)):
        if currentMax < A[i]:
            currentMax = A[i]
    return currentMax
```

... more python-ish

- Algorithm for finding the maximum element of an array

```python
def iMax(A):
    currentMax = A[0]
    for x in A[1:]:
        if currentMax < x:
            currentMax = x
    return currentMax
```
Input size and basic operation examples

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input size measure</th>
<th>Basic operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Searching for key in a list of $n$ items</td>
<td>Number of items in the list, i.e., $n$</td>
<td>Key comparison</td>
</tr>
<tr>
<td>Multiplication of two matrices</td>
<td>Matrix dimensions or total number of elements</td>
<td>Multiplication of two numbers</td>
</tr>
<tr>
<td>Checking primality of a given integer $n$</td>
<td>size of $n$ = number of digits (in binary representation)</td>
<td>Division</td>
</tr>
<tr>
<td>Typical graph problem</td>
<td>#vertices and/or #edges</td>
<td>Visiting a vertex or traversing an edge</td>
</tr>
</tbody>
</table>

Example (Max iterative)

```python
def iMax(A):
    currentMax = A[0]
    for i in range(len(A)):
        if currentMax < A[i]:
            currentMax = A[i]
    return currentMax
```

The program executes $n-1$ comparisons (irrespective from the type of input) where $n=len(A)$ therefore the worst case time-complexity is $O(n)$
Example (Max recursive)

```python
def rMax(A):
    if len(A) == 1:
        return A[0]
    return max(rMax(A[1:]), A[0])
```

The program executes $n-1$ comparisons (irrespective from the type of input) therefore the worst case time-complexity is $O(n)$

Asymptotic notation

Section 0.3 of the textbook
The “Big-Oh” Notation

- **Definition:** Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if and only if there are positive constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$.

Asymptotic Notation

- **Special classes of algorithms**
  - constant: $O(1)$
  - logarithmic: $O(\log n)$
  - linear: $O(n)$
  - quadratic: $O(n^2)$
  - cubic: $O(n^3)$
  - polynomial: $O(n^k)$, $k \geq 1$
  - exponential: $O(a^n)$, $n > 1$
Asymptotic Notation

• “Relatives” of the Big-Oh
  – $\Omega(f(n))$: Big Omega
    • asymptotic lower bound
  – $\Theta(f(n))$: Big Theta
    • asymptotic tight bound

Big Omega

• Definition: Given two functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Omega(g(n))$
  if and only if
  there are positive constants $c$ and $n_0$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

• Property: $f(n)$ is $\Omega(g(n))$ iff $g(n)$ is $O(f(n))$
Big Theta

- **Definition:** Given two functions \( f(n) \) and \( g(n) \), we say that \( f(n) \) is \( \Theta(g(n)) \) if and only if there are positive constants \( c_1, c_2 \) and \( n_0 \) such that \( c_1 g(n) \leq f(n) \leq c_2 g(n) \) for \( n \geq n_0 \).

- **Property:** \( f(n) \) is \( \Theta(g(n)) \) if and only if “\( f(n) \) is \( O(g(n)) \) AND \( f(n) \) is \( \Omega(g(n)) \)”

Establishing order of growth using limits

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} 
0 & \text{order of growth of } f(n) < \text{order of growth of } g(n) \\
c > 0 & \text{order of growth of } f(n) = \text{order of growth of } g(n) \\
\infty & \text{order of growth of } f(n) > \text{order of growth of } g(n)
\end{cases}
\]

**Examples:**

- \( 10n \) vs. \( n^2 \)
- \( n(n+1)/2 \) vs. \( n^2 \)
Orders of growth: some important functions

• All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ no matter what the logarithm’s base $a > 1$ is.
• All polynomials of the same degree $k$ belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$ in $\Theta(n^k)$.
• Exponential functions $a^n$ have different orders of growth for different $a$’s.
• order $\log n < \text{order } n < \text{order } n \log n < \text{order } n^k$ ($k \geq 2$ constant) $< \text{order } a^n < \text{order } n! < \text{order } n^n$.
• Caution: Be aware of very large constant factors.

Suppose each operation takes 1 nanoseconds ($10^{-9}$ seconds)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log n$</th>
<th>$n$</th>
<th>$\log n$</th>
<th>$n^2$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.003μs</td>
<td>0.01μs</td>
<td>0.033μs</td>
<td>0.1μs</td>
<td>1μs</td>
<td>3.63ms</td>
</tr>
<tr>
<td>20</td>
<td>0.004μs</td>
<td>0.02μs</td>
<td>0.086μs</td>
<td>0.4μs</td>
<td>1ms</td>
<td>77.1years</td>
</tr>
<tr>
<td>30</td>
<td>0.005μs</td>
<td>0.02μs</td>
<td>0.147μs</td>
<td>0.9μs</td>
<td>1sec</td>
<td>&gt;10^13years</td>
</tr>
<tr>
<td>100</td>
<td>0.007μs</td>
<td>0.1μs</td>
<td>0.644μs</td>
<td>10μs</td>
<td>&gt;10^13years</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>0.013μs</td>
<td>10μs</td>
<td>130μs</td>
<td>100ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.020μs</td>
<td>1ms</td>
<td>19.92μs</td>
<td>16.7min</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• For $n < 10$, the difference is insignificant.
• $\Theta(n!)$ algorithms are useless well before $n = 20$.
• $\Theta(2^n)$ algorithms are practical for $n < 40$.
• $\Theta(n^2)$ and $\Theta(n \log n)$ are both useful, but $\Theta(n \log n)$ is significantly faster.
Time analysis for iterative algorithms

Steps

• Decide on parameter $n$ indicating input size
• Identify algorithm’s basic operation
• Determine worst case(s) for input of size $n$
• Set up a sum for the number of times the basic operation is executed
• Simplify the sum using standard formulas and rules

Example of Asymptotic Analysis

```python
def prefixAverages1(X):
    A = []
    for i in range(len(X)):
        a = 0
        for j in range(i+1):
            a += X[j]
        A.append(a/float(i+1))
    return A
```

...then the algorithm is $O(n^2)$
A faster algorithm

• Observe that

\[
\begin{align*}
A[i-1] &= (X[0] + X[1] + \cdots + X[i-1])/i \\
A[i] &= (X[0] + X[1] + \cdots + X[i-1] + X[i])/(i+1).
\end{align*}
\]

A linear-time algorithm

```python
def prefixAverages2(X):
    A, a = [], 0
    for i in range(len(X)):
        a = a + X[i]
        A.append(a/float(i+1))
    return A
```
A trickier example

• Analyze the worst-case time complexity of the following algorithm, and give a tight bound using the big-theta notation

```python
def weirdLoop(n):
    i = n
    while i >= 1:
        for j in range(i):
            print 'Hello'
        i = i/2
    return
```

Math review

Appendix A of the textbook
Summations

\[ \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \]
\[ \sum_{i=0}^{n} a^i = \frac{1 - a^{n+1}}{1-a} \text{ when } a > 0, \ a \neq 1 \]
\[ \text{e.g., } \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + ... + 2^n = 2^{n+1} - 1 \]

Bounding sums

- **Upper bound:** Any sum is at most the number of terms times the maximum term
  - Example: \(1+4+9+...+n^2\) is at most \(n^2n^2 = n^3\)
- **Lower bound:** If the terms are non-negative, any sum is at least half the number of terms times the median term
  - Example: \(1+4+9+...+n^2\) is at least \((n/2)^2(n/2)^2 = n^3/8\)
Proving (or disproving) $p \rightarrow q$

- **Counterexample** (used to prove that $p \rightarrow q$ is false showing one particular choice of $p$ that makes $q$ false)
- **Direct** proof ($p \rightarrow p_1 \rightarrow \ldots \rightarrow p_n \rightarrow q$)
- **Contrapositive** (prove that $\neg q \rightarrow \neg p$)
- **Contradiction** (assume $p$ and $\neg q$ true, find a contradiction)
- **Induction** (prove base case + induction)

\[ \begin{align*}
\text{Theorem:} & \quad \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \\
\text{Proof:} & \quad \text{by induction on } n. \\
\text{Base case:} & \quad n = 1. \text{ Trivial since } 1 = 1(1+1)/2. \\
\text{Induction step:} & \quad n \geq 2. \text{ Assume the claim is true for any } n' < n. \text{ Then } \\
& \quad \sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i = n + \frac{(n-1)n}{2} = \frac{n(n+1)}{2} \\
& \quad \text{using induction}
\end{align*} \]
Recurrence Relation Analysis

Recurrence relation

• A recurrence relation is an equation that recursively define a sequence: each term of the sequence is defined as a function of the preceding term(s)
• For instance
  \[
  f(n) = \begin{cases} 
  2 & n=1 \\
  f(n-1) + n & n>1 
  \end{cases}
  \]
General form

$$T(n) = \begin{cases} 
    c & \text{if } n = n_0 \\
    a.T(f(n)) + g(n) & \text{otherwise}
\end{cases}$$

MergeSort

- MergeSort is a divide & conquer algorithm
  - Divide: divide an $n$-element sequence into two subsequences of approx $n/2$ elements
  - Conquer: sort the subsequences recursively
  - Combine: merge the two sorted subsequences to produce the final sorted sequence
MergeSort

```python
def mergesort(A):
    if len(A) < 2:
        return A
    else:
        m = len(A)/2
        l = mergesort(A[:m])
        r = mergesort(A[m:])
        return merge(l, r)
```

Example

![Figure 4.2: Merge-sort tree $T$ for an execution of the merge-sort algorithm on a sequence with 8 elements: (a) input sequences processed at each node of $T$; (b) output sequences generated at each node of $T.$](image)

Stefano Lonardi, UCR
Merge of MergeSort

```python
def merge(l, r):
    result, i, j = [], 0, 0
    while i < len(l) and j < len(r):
        if l[i] <= r[j]:
            result.append(l[i])
            i += 1
        else:
            result.append(r[j])
            j += 1
    result += l[i:]
    result += r[j:]
    return result
```

MergeSort Analysis

- **Divide:** Just computes the middle of the subsequence, thus takes constant time:
  \[ D(n) = \Theta(1) \]

- **Conquer:** We solve 2 subproblems of size approximately \( n/2 \):
  \[ a = 2, \quad b = 2 \]

- **Combine:** Merge takes \( \Theta(n) \):
  \[ C(n) = \Theta(n) \]

- Noting that \( \Theta(n) + \Theta(1) \) is still \( \Theta(n) \), we get:
  \[
  T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2 \ T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}
  \]

- Later we will see that:
  \[ T(n) = \Theta(n \lg n) \]
“Visual” Analysis

Figure 4.4: A visual analysis of the running time of merge-sort. Each node of the merge-sort tree is labeled with the size of its subproblem.

Solving Recurrence Relation
Methods

- Two methods for solving recurrences
  - Iterative substitution method
  - Master method

  - (not covered: Recursion Tree)
  - (not covered: Guess-and-Test method)

Iterative substitution

- Assume \( n \) large enough
- Substitute \( T \) on the right-hand side of the recurrence relation
- Iterate the substitution until we see a pattern which can be converted into a general closed-form formula
MergeSort recurrence relation

\[ T(N) = 2T\left(\frac{N}{2}\right) + N \quad \text{for} \quad N \geq 2 \]
\[ T(1) = 1 \]

\[ T(N) = 2\left(2T\left(\frac{N}{4}\right) + \frac{N}{2}\right) + N \]
\[ = 4T\left(\frac{N}{4}\right) + 2N \]
\[ = 4\left(2T\left(\frac{N}{8}\right) + \frac{N}{4}\right) + 2N \]
\[ = 8T\left(\frac{N}{8}\right) + 3N \]
\[ \vdots \]
\[ = 2^i T\left(\frac{N}{2^i}\right) + iN \]

The expansion stops for \( i = \log_2 N \), so that
\[ T(N) = N + N \log_2 N \]
Verify the correctness

- How to verify the solution is correct?
- Use proof by induction!
- Important: make sure the constant $c$ works for both the base case and the induction step

Proof by induction

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Fact: $T(n) \in O(n \log_2 n)$

Proof. Base case: $T(2) = 2T(1) + 2 = 4 \leq c(2 \log_2 2) = 2c$. Hence, $c \geq 2$.

Induction hypothesis: $T(n/2) \leq c \frac{n}{2} \log_2 \frac{n}{2}$

Induction: $T(n) = 2T(n/2) + n$

\[ \leq 2c \frac{n}{2} \log_2 \frac{n}{2} + n \]
\[ = cn \log_2 \frac{n}{2} + n = cn \log_2 n - cn \log_2 2 + n \]
\[ = cn \log_2 n + n(1-c) \leq cn \log_2 n \text{ when } c \geq 1 \]

Choose $c = 2$. 

The constant $c$ used in the induction and the base case has to be the same!
Wrong proof by induction

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise} 
\end{cases} \]

Fact (wrong): \( T(n) \in O(n) \)

Proof. Base case: \( T(1) = 1 \leq c1 \), hence \( c \geq 1 \)

Induction hypothesis: \( T(n/2) \leq c(n/2) \)

Induction: \( T(n) = 2T(n/2) + n \)
\[
\leq 2c(n/2) + n \\
= cn + n \in O(n)
\]

proof is WRONG, but where is the mistake?

Towers of Hanoi

![Towers of Hanoi diagram]
Towers of Hanoi

**Goal:** transfer all $N$ disks from peg A to peg C

**Rules:**
- move one disk at a time
- never place larger disk above smaller one

**Recursive solution:**
- transfer $N-1$ disks from A to B
- move largest disk from A to C
- transfer $N-1$ disks from B to C

**Total number of moves:**
- $T(N) = 2T(N-1) + 1$

---

def hanoi(n, a='A', b='B', c='C'):
    if n == 0:
        return
    hanoi(n-1, a, c, b)
    print a, '->', c
    hanoi(n-1, b, a, c)
Towers of Hanoi: Recurrence Relation

Solve

\[ T(N) = \begin{cases} 
2T(N-1) + 1 & N > 1 \\
1 & N = 1 
\end{cases} \]

Towers of Hanoi: Unfolding the relation

\[ T(N) = 2 \left( 2 \left( T(N-2) + 1 \right) + 1 \right) = \]
\[ = 4 \, T(N-2) + 2 + 1 = \]
\[ = 4 \left( 2 \, T(N-3) + 1 \right) + 2 + 1 = \]
\[ = 8 \, T(N-3) + 4 + 2 + 1 = \]
\[ \ldots \]
\[ = 2^i \, T(N-i) + 2^{i-1} + 2^{i-2} + \ldots + 2^1 + 2^0 \]

the expansion stops when \( i = N - 1 \)

\[ T(N) = 2^{N-1} + 2^{N-2} + 2^{N-3} + \ldots + 2^1 + 2^0 \]

This is a geometric sum, so that we have:

\[ T(N) = 2^N - 1 \in \Theta(2^N) \]
Problem

Problem: Solve exactly (by iterative substitution)

\[ T(n) = \begin{cases} 
4 & n = 1 \\
4T(n-1) + 3 & n > 1 
\end{cases} \]

Solution: \( T(n) = 4^n + 4^{n-1} - 1 \)

Proof?
Another example

\[ T(N) = 2T(\sqrt{N}) + 1 \quad \quad T(2) = 0 \]

\[ 2T(N^{1/2}) + 1 \]
\[ 2(2T(N^{1/4}) + 1) + 1 \]
\[ 4T(N^{1/4}) + 1 + 2 \]
\[ 8T(N^{1/8}) + 1 + 2 + 4 \]
\[ \ldots \]

Another example

\[ 2^iT \left( \frac{1}{N^{2^i}} \right) + 2^0 + 2^1 + \ldots + 2^{i-1} \]

The expansion stops for \( \frac{1}{N^{2^i}} = 2 \)
i.e., \( i = \log \log N \)

\[ T(N) = 2^0 + 2^1 + \ldots + 2^{\log \log N - 1} = \log N - 1 \]
Master Theorem method

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d
\end{cases} \]

**Theorem 5.6 [The Master Theorem]:** Let \( f(n) \) and \( T(n) \) be defined as above.

1. If there is a small constant \( \varepsilon > 0 \) such that \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \).
2. If there is a constant \( k \geq 0 \) such that \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \).
3. If there are small constants \( \varepsilon > 0 \) and \( \delta < 1 \) such that \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \) and \( af(n/b) \leq \delta f(n) \), for \( n \geq d \), then \( T(n) \) is \( \Theta(f(n)) \).

\( \frac{n}{b} \) stands for \( \lfloor n/b \rfloor \) or \( \lceil n/b \rceil \)
Master method (first case)

Example 5.7: Consider the recurrence
\[ T(n) = 4T(n/2) + n. \]
In this case, \( n^{\log_2 4} = n^2 \). Thus, we are in Case 1, for \( f(n) = O(n^{2-\varepsilon}) \) for \( \varepsilon = 1 \). This means that \( T(n) \) is \( \Theta(n^2) \) by the master method.

Master method (second case)

Example 5.8: Consider the recurrence
\[ T(n) = 2T(n/2) + n\log n, \]
which is one of the recurrences given above. In this case, \( n^{\log_2 2} = n \). Thus, we are in Case 2, with \( k = 1 \), for \( f(n) = \Theta(n\log n) \). This means that \( T(n) \) is \( \Theta(n\log^2 n) \) by the master method.
Master method: binary search (second case)

- The Master Theorem allows us to ignore the floor or ceiling function around $n/b$ in $T(n/b)$ in general.
- Binary Search has for any $n > 0$ a running time of
  \[ T(n) = T(n/2) + \Theta(1) \]
  Hence $a = 1$, $b = 2$, $f(n) = \Theta(1)$. Since $1 = n^{\log_2 2}$ the second case applies and we get:
  \[ T(n) = \Theta(\lg n) \]

Master method: merge-sort (second case)

- For arbitrary $n > 0$, the running time of Merge-Sort is
  \[
  T(n) = \begin{cases} 
  \Theta(1) & \text{if } n = 1 \\
  T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 
  \end{cases}
  \]
  We can approximate this from below and above by
  \[
  T(n) = \begin{cases} 
  2T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 \\
  2T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 
  \end{cases}
  \]
  respectively. According to the Master Theorem, both have the same solution which we get by taking
  \[ a = 2, \ b = 2, \ f(n) = \Theta(n). \]
  Since $n = n^{\log_2 2}$, the second case applies and we get:
  \[ T(n) = \Theta(n \lg n) \]
Master method (third case)

Example 5.9: Consider the recurrence

\[ T(n) = T(n/3) + n, \]

which is the recurrence for a geometrically decreasing summation that starts with \( n \).

In this case, \( n^{\log_3 3} = n^1 = n^0 = n \). Thus, we are in Case 3, for \( f(n) \) is \( \Omega(n^{0+\varepsilon}) \), for \( \varepsilon = 1 \), and \( af(n/b) = n/3 = (1/3)f(n) \). This means that \( T(n) \) is \( \Theta(n) \) by the master method.

Example 5.10: Consider the recurrence

\[ T(n) = 9T(n/3) + n^{2.5}. \]

In this case, \( n^{\log_3 9} = n^2 \). Thus, we are in Case 3, for \( f(n) \) is \( \Omega(n^{2+\varepsilon}) \), for \( \varepsilon = 1/2 \), and \( af(n/b) = 9(n/3)^{2.5} = (1/3)^{1/2}f(n) \). This means that \( T(n) \) is \( \Theta(n^{2.5}) \) by the master method.

Summary (1/3)

- **Goal:** analyze the worst-case time-complexity of iterative and recursive algorithms
- **Tools:**
  - Pseudo-code/Python
  - Big-O, Big-Omega, Big-Theta notations
  - Recurrence relations
  - Discrete Math (summations, induction proofs, methods to solve recurrence relations)
Summary (2/3)

• Pure iterative algorithm:
  – Analyze the loops
  – Determine how many times the inner core is repeated as a function of the input size
  – Determine the worst-case for the input
  – Write the number of repetitions as a function of the input size
  – Simplify the function using big-O or big-Theta notation (optional)

Summary (3/3)

• Recursive + iterative algorithm:
  – Analyze the recursive calls and the loops
  – Determine how many recursive calls are made and the size of the arguments of the recursive calls
  – Determine how much extra processing (loops) is done
  – Determine the worst-case for the input
  – Derive a recurrence relation
  – Solve the recurrence relation
  – Simplify the solution using big-O, or big-Theta