Analysis of Algorithms

Outline

- Worst case time-complexity
- Asymptotic notation
- Lower bounds
- Discrete Math & Recurrence Relations
- Amortized Analysis
Various algorithmic “complexities”

• We can “rank” algorithms depending on several factors
  – running time (“time complexity”)
  – memory requirements (“space complexity”)
  – power consumption
  – 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 input size*, when the input is the one that makes the algorithm *slower* in the limit

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

Usage of Python

• We will use Python code (when possible) to describe algorithms (sometimes w English)

• Python is
  – High-level (easy to use and 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(1,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
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 $n \geq n_0$.

Figure 1.3: Illustrating the “big-Oh” notation. The function $f(n)$ is $O(g(n))$, for $f(n) \leq c \cdot g(n)$ when $n \geq n_0$. 
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: \( O(n^k), k \geq 0 \)
  - exponential: \( O(a^n), n > 1 \)

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 \( \Omega(g(n)) \)”

Asymptotic Analysis of Running Time

- **Comparing the asymptotic running time**
  - an algorithm that runs in \( O(n) \) time is **better** than one that runs in \( O(n^2) \) time
  - similarly, \( O(\log n) \) is **better** than \( O(n) \)
  - hierarchy of functions: \( \log n < n < n^2 < n^3 < 2^n \)

- **Caution**: Beware of very large constant factors. An algorithm running in time \( 1,000,000 \ n \) is still \( O(n) \) but might be less efficient on your data set than one running in time \( 2n^2 \), which is \( O(n^2) \)
Time analysis for iterative algorithms

Steps

1. Decide on parameter \( n \) indicating input size
2. Identify algorithm’s basic operation
3. Determine worst case(s) for input of size \( n \)
4. Set up a sum for the number of times the basic operation is executed
5. 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]
            # n iterations
        A.append(a/float(i+1))
    return A
```

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

A faster algorithm

• Observe that

\[
A[i-1] = \frac{X[0] + X[1] + \cdots + X[i-1]}{i} \\
A[i] = \frac{X[0] + X[1] + \cdots + X[i-1] + X[i]}{i+1}.
\]
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
```
Lower bounds

Lower bounds: intro

• Most of the class will be devoted to solve certain problems as quickly as possible
• By showing faster and faster algorithms for a specific problem, are making statements on how *easy* the problem is
• Sometimes we are interested to show how *hard* some problems are by proving lower bounds on their complexity
Lower bounds: intro

- This is considerably harder than proving upper bounds because it is no longer enough to examine a single algorithm.
- To prove that a problem $P$ cannot be solved faster than $f(n)$ time for an input of size $n$, we must prove that every algorithm that solves $P$ has a worst-case running time $\Omega(f(n))$.

Decision-tree model of computation

- Many sorting and searching algorithms are comparison-based, i.e., they sort/search by making comparisons between pairs of objects (examples: binary search, bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, …).

- We define the running time of a decision tree algorithm for a given input to be the number of queries in the path from the root to the leaf.
Decision-tree model

- Each internal node is a query (question about the input), edges represent possible answers (constant), each leaf is labeled with a possible output
- Search: suppose we want to determine, given a number $x$, the position of $x$ in the array $A$, if any
- The binary search tree is an implicit decision-tree model

Comparison-based search

- Most lower bounds for decision trees are based on the following simple observation: \textit{the answers to the queries must give you enough information to specify any possible output}
- If a problem has $N$ different outputs, then any decision tree must have at least $N$ leaves
- If every query has two possible answers, the height of the decision tree must be at least $\Omega(\log N)$
- In the search problem, there are $n+1$ possible outputs, any decision tree must have at least $n+1$ leaves, and thus any decision tree must have depth at least $\Omega(\log n)$
- This implies that the standard binary search algorithm is optimal (there is no faster algorithm in this model of computation)
Comparison-based sorting

- Let us derive a lower bound on the running time of any algorithm that uses comparisons to sort \( n \) elements \( x_1, x_2, ..., x_n \) (distinct) by counting the number of comparisons
- Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree

\[ \begin{align*}
&x_i < x_j \quad \text{?} \\
&x_a < x_b \quad \text{?} \\
&x_m < x_o \quad \text{?} \\
&x_p < x_q \quad \text{?} \\
&x_e < x_f \quad \text{?} \\
&x_k < x_l \quad \text{?} \\
&x_c < x_d \quad \text{?}
\end{align*} \]

**Decision Tree Height**

- The height of this decision tree is a lower bound on the running time
- Every possible input permutation must lead to a separate leaf output
- Since there are \( n! \) leaves, the height is at least \( \log(n!) \)
The Lower Bound

• Any comparison-based sorting algorithms takes at least \( \log(n!) \) time
• Because of the Stirling’s approximation
  \[
  n! \approx n^n e^{-n} \sqrt{2\pi n}
  \]
  \[
  \ln(n!) \approx n \ln n - n
  \]
• Thus, any comparison-based sorting algorithm must run in \( \Omega(n \log n) \) time in the worst case
• Decision-tree describe almost all well-known sorting algorithms where the actual input values don’t matter, only their order
• This lower bound does not apply to numbers in a fixed range (e.g., integers) one can sort faster (counting/bucket/radix sort)

Recurrence Relations
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

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

Recurrence relations: simple form
MergeSort: sorting recursively

- 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

```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**

![Merge-sort tree](image)

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 $T$.

---

**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
```

44
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:
  \[ T(n) = \Theta(1) \quad \text{if} \quad n = 1 \]
  \[ 2T(n/2) + \Theta(n) \quad \text{if} \quad n > 1 \]
- Later we will see that:
  \[ T(n) = \Theta(n \log n) \]

“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 relations: Methods

- Two methods for solving recurrences
  - Iterative substitution method
  - Master method
    - (Recursion Tree)
    - (Guess-and-Test method)

Mergesort recurrence relation

\[
\begin{align*}
T(N) &= 2T\left(\frac{N}{2}\right) + N & \text{for } N \geq 2 \\
T(1) &= 1
\end{align*}
\]
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

$$T(N) = 2 \left( 2T \left( \frac{N}{4} \right) + \frac{N}{2} \right) + N$$

$$= 4T \left( \frac{N}{4} \right) + 2N$$

$$= 4 \left( 2T \left( \frac{N}{8} \right) + \frac{N}{4} \right) + 2N$$

$$= 8T \left( \frac{N}{8} \right) + 3N$$

$$\ldots$$

$$= 2^{i}T \left( \frac{N}{2^i} \right) + iN$$

The expansion stops for $i = \log_2 N$, so that

$$T(N) = N + N \log_2 N$$
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: \( T(n) = 2T(n/2) + n \)

\[ \leq 2c \frac{n}{2} \log_2 \frac{n}{2} + n \]

\[ = cn \log_2 \frac{n}{2} + n = cn \log_2 n - cn \log_2 2 + n \]

Choose \( c = 2 \).

The constant \( c \) used in the induction and the base case has to be the same!

Towers of Hanoi
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:**
- $T(N) = 2T(N-1) + 1$

```python
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

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

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

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

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

Another example

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

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 \( aT(n/b) \leq \delta f(n) \), for \( n \geq d \), then \( T(n) \) is \( \Theta(f(n)) \).

\( n/b \) stands for \( \lceil n/b \rceil \) or \( \lfloor n/b \rfloor \)

---

**Master Theorem**

<table>
<thead>
<tr>
<th>Condition on ( f(n) )</th>
<th>Condition</th>
<th>Conclusion on ( T(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(n^{\log_b a - \varepsilon}) )</td>
<td>( \varepsilon &gt; 0 )</td>
<td>( \Theta(n^{\log_b a}) )</td>
</tr>
<tr>
<td>( \Theta(n^{\log_b a \log^k n}) )</td>
<td>( k \geq 0 )</td>
<td>( \Theta(n^{\log_b a \log^{k+1} n}) )</td>
</tr>
<tr>
<td>( \Omega(n^{\log_b a + \varepsilon}) )</td>
<td>( \varepsilon &gt; 0, \delta &lt; 1 ) ( af(n/b) \leq \delta f(n) )</td>
<td>( \Theta(f(n)) )</td>
</tr>
</tbody>
</table>
Master method (first case)

Example 5.7: Consider the recurrence

\[ T(n) = 4T(n/2) + n. \]

In this case, \( n^{\log_2 a} = n^{\log_2 4} = n^2 \). Thus, we are in Case 1, for \( f(n) \) is \( O(n^{2-\varepsilon}) \) for \( \varepsilon = 1 \). This means that \( T(n) \) is \( \Theta(n^2) \) by the master method.

---

Master method: Hanoi (first case w substitution)

- Hanoi has for any \( n > 0 \) a running time of
  \[ T(n) = 2T(n-1) + 1. \]

  In order to bring this into a form such that the Master Theorem is applicable, we rename \( n = \lg m \):
  \[
  T(\lg m) = 2T(\lg m - 1) + 1 \\
  = 2T(\lg m - \lg 2) + 1 \\
  = 2T(\lg (m/2)) + 1
  \]

  Defining \( S(m) = T(\lg m) \) we get the new recurrence:
  \( S(m) = 2S(m/2) + 1 \)

  Hence \( a = 2, b = 2, f(m) = 1 \). Since \( 1 = m^{\log_2 2} \) the first case applies with \( \varepsilon = 1 \) and we get:
  \( S(m) = \Theta(m) \)

  With \( S(m) = T(\lg m) \) and \( n = \lg m \) we finally get:
  \( T(n) = \Theta(2^n) \)
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, \( n^{\log_2 1} = n^{\log_2 2} = n \).
Thus, we are in Case 2, with \( k = 1 \), for \( f(n) \) is \( \Theta(n \log n) \). 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 \( 1 = n^{\log_2 1} \) the second case applies and we get:

\[ T(n) = \Theta(\log n) \]
Master method: merge-sort (second case)

- For arbitrary $n > 0$, the running time of Merge-Sort is
  \[
  T(n) = \begin{cases} 
  O(1) & \text{if } n = 1 \\
  T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1
  \end{cases}
  \]

  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:
  \[T(n) = \Theta(n \log n)\]

Master method (second case w substitution)

**Example 5.11:** Finally, consider the recurrence

\[ T(n) = 2T(n^{1/2}) + \log n. \]

This equation is unfortunately not in a form that allows us to use the master method. We can put it into such a form, however, by introducing the variable $k = \log n$, which lets us write

\[ T(n) = T(2^k) = 2T(2^{k/2}) + k. \]

Substituting into this the equation $S(k) = T(2^k)$, we get that

\[ S(k) = 2S(k/2) + k. \]

Now, this recurrence equation allows us to use master method, which specifies that $S(k)$ is $O(k \log k)$. Substituting back for $T(n)$ implies $T(n)$ is $O(\log n \log \log n)$. 
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, \( n^{\log_3 a} = n^{\log_3 1} = n^0 = 1 \). Thus, we are in Case 3, for \( f(n) \) is \( \Omega(n^{0+\varepsilon}) \),
for \( \varepsilon = 1 \), and \( a f(n/b) = n/3 = (1/3)f(n) \). 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, \( n^{\log_3 a} = n^{\log_3 9} = n^2 \). Thus, we are in Case 3, for \( f(n) \) is \( \Omega(n^{2+\varepsilon}) \), for
\( \varepsilon = 1/2 \), and \( a f(n/b) = 9(n/3)^{2.5} = (1/3)^{1/2} f(n) \). This means that \( T(n) \) is \( \Theta(n^{2.5}) \)
by the master method.

Amortized analysis
Amortized Analysis

• In *amortized analysis* we care for the temporal cost of one operation when considered *in an aggregate sequence of n operations*

• In a sequence of *n* operations, some operations may be cheap, some may be expensive

• The *amortized cost* of an operation equals the *total* cost of the *n* operations divided by *n*

Amortized Analysis

• The goal is to find an upper bound for the total time complexity *T(n)* required for a sequence of *n* operations

• Formally: if your algorithm takes a total *T(n)* time/work for *n* operations, then the *amortized cost* of each operation is  *T(n)/n*
Amortized Analysis: Outline

- We will prove amortized run times using the *accounting method*.
- We present how the accounting works with two examples:
  - Stack example
  - Binary counter example
- Other amortized techniques:
  - Aggregate analysis
  - Potential method

Amortized Analysis: Stack Example

Consider a stack $S$ that holds up to $n$ elements and it has the following three operations:

- $\text{PUSH}(S, x)$  ... pushes object $x$ in stack $S$
- $\text{POP}(S)$  ... pops top of stack $S$
- $\text{MULTIPOP}(S, k)$ ... pops the $k$ top elements of $S$ or pops the entire stack if it has less than $k$ elements
Amortized Analysis: Stack Example

• How much a sequence of $n$ \texttt{PUSH()}, \texttt{POP()} and \texttt{MULTIPOP()} operations cost?
  – A \texttt{MULTIPOP()} may take $O(n)$ time in the worst-case
  – \textit{Naïve} analysis: a sequence of $n$ such operations may take $O(n \times n) = O(n^2)$ time since we may call $n$ \texttt{MULTIPOP()} operations of $O(n)$ time each

• With \textit{accounting method} (amortized analysis) we can show a better run time of $O(1)$ per operation

Amortized Analysis: Stack Example

• Devise a charging scheme that \textit{accounts} for the time of a basic operation (in this case, either push or pop of one element)

• Proposed charging scheme
  – Charge $\$2$ for operation \texttt{PUSH()}
    • $\$1$ pays for the cost of pushing an element
    • $\$1$ is deposited on the element to pay when/if POP-ed later by either \texttt{POP()} or \texttt{MULTIPOP()}
  – Charge $\$0$ for \texttt{POP()} and a \texttt{MULTIPOP()}

75

76
Amortized Analysis: Stack Example

\[ \text{credit invariant} \]

\[ \begin{array}{c}
\text{Push (a)} = 2 \\
\text{Push (b)} = 2 \\
\text{Push (c)} = 2 \\
\end{array} \]

- $1$ pays for push and $1$ is deposited
- $1$ pays for push and $1$ is deposited
- $1$ pays for push and $1$ is deposited

\[ \begin{array}{c}
\text{MULTIPOP () costs nothing} \\
because you have the $1$ bills to pay for the pop operations!
\end{array} \]

Accounting Method

- We operate with a budget \( T(n) \)
  - A sequence of \( n \) \texttt{POP()}, \texttt{MULTIPOP()}, and \texttt{PUSH()} operations needs a budget \( T(n) \) of at most $2n$
  - Each operation costs

\[ T(n)/n = 2n/n = O(1) \text{ amortized time} \]
Binary Counter Example

• Let $A$ be a $n$-bit counter $A[n-1]...A[0]$ (counts from 0 to $2^n-1$)
  – How much time does it take to increment the counter $n$ times starting from zero?
  – We want to measure the number $T(n)$ of bits we need to flip ($0 \to 1$ and $1 \to 0$) as we increment the $n$-bit counter (time complexity)

```python
def increment(A):
    i=0
    while i<len(A) and A[i]==1:
        A[i]=0
        i+=1
    if i<len(A):
        A[i]=1
    return A
```

This procedure resets the first $i$-th sequence of 1 bits and sets $A[i]$ equal to 1 (ex. $0011 \to 0100$, $0101 \to 0110$, $1111 \to 0000$)
### 4-bit Binary Counter Example

<table>
<thead>
<tr>
<th>Counter value</th>
<th>COUNTER</th>
<th>Bits flipped (work $T(n)$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>15</td>
</tr>
</tbody>
</table>

Highlighted are bits that flip at each increment

---

### Binary Counter Example

- A naïve analysis would show that a sequence of $n$ operations on a $n$-bit counter needs $O(n^2)$ work
  - Each `INCREMENT()` takes up to $O(n)$ time $\rightarrow n$ `INCREMENT()` operations can take $O(n^2)$ time

- Amortized analysis with accounting method
  - We show that amortized cost per `INCREMENT()` is only $O(1)$ and the total work $O(n)$
  - Aggregate analysis: Prove that $T(n)$ is never twice the amount of counter value (total # of increments)
Binary Counter Example

- Charge $0 for $A[i]=0$
- Charge $2$ for $A[i]=1$
  - $1$ pays for the 0→1 flip in $A[i]=1$
  - $1$ is deposited to pay for the 1→0 flip later in $A[i]=0$
- Therefore, a sequence of $n$ INCREASES () needs
  \[ T(n) = 2n \]
  …each INCREMENT () has an amortized cost of
  \[ \frac{2n}{n} = O(1) \]
Reading assignment

• Chapter 3, “Growth of Functions”
• Section 8.1, “Lower bounds for sorting”
• Chapter 4, “Recurrences”
• Chapter 17, “Amortized Analysis”