Advanced Algorithms Analysis and Design

Lecture 8

Analyzing loops and recursive functions

Dr. Shamila Nasreen

Today’s Lectures

- Iterative Algorithms: Analyzing Loops
- Few Analysis Examples
- Analysis of Control Structure
- Recursive calls
- While and Repeat Loop
- Recurrence Relation

Iterative Algorithms: Analyzing Loops

- An iterative algorithm uses **loops** (e.g., for, while) to repeat tasks. To find its time complexity, we count the total number of operations by analyzing the loops.

Iterative Algorithms: Simple Loop

- Consider this code to sum an array:

```python
def sum_array(arr):
    total = 0
    for x in arr:
        total += x    # O(1) operation
    return total
```

Step by Step Cost analysis:

Line	Statement	Cost
1	`total = 0`	c_1
2	`for x in arr`	$c_2 \times (n + 1)$
3	`total += x`	$c_3 \times n$
4	`return total`	c_4

Final Total Cost Expression:

$$\boxed{c_1 + c_2 \times (n+1) + c_3 \times n + c_4}$$

Asymptotic Complexity (Big-O):

Dominated by the linear terms (terms multiplied by **n**).

So, overall **Time Complexity = O(n)**.

Important Summations

Arithmetic series

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$
$$= \frac{n(n+1)}{2} = \Theta(n^2)$$

Quadratic series

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \ldots + n^2$$
$$= \frac{2n^3 + 3n^2 + n}{6} = \Theta(n^3)$$

Geometric series

$$\sum_{i=1}^{n} x^i = 1 + x + x^2 + \ldots + x^n$$
$$= \frac{x^{(n+1)} - 1}{x - 1} = \Theta(n^2)$$

Important Summations that should be Committed to Memory.

Harmonic series For $n \geq 0$

$$
\begin{aligned}
H_n &= \sum_{i=1}^{n} \frac{1}{i} \\
&= 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n \\
&= \Theta(\ln n)
\end{aligned}
$$

Analyzing Control Structures "For" loops

- Consider the loop
- **for** i $\leftarrow$ 1 **to** m **do** P(i)

- Suppose the loop is part of a larger algorithm working on an instance of size *n.* Let *t* denote the time required to compute *P(i)*

- *P(i)* is performed *m* times, each time at a cost of *t* Total time required by the loop is l = *mt*

If the time *t(i)* required for *P(i)* varies as a function of *i*, the loop takes a time given by the sum

$$\sum_{i=1}^{m} t(i)$$

Analysing Control Structures
"For" loops

for i ← 1 **to** m **do** P(i)

* Time needed for loop control must also be taken into account. For loop can be like:

$i \leftarrow 1$	: c
while $i \leq m$ do	: (m+1)c
P(i)	: mt
$i \leftarrow i + 1$	: mc
for sequencing operations(go to)	: mc

$l \leq (c + (m+1)c+mt+mc+mc)$

$l \leq (t+3c)m+2c)$

This *jumping back* (sequencing the control flow) **also takes time — it's called a sequencing operation** or **"go to" operation**.

Analysing Control Structures

"For" loops

* Where c can be upper bound on the time required by each of the above mentioned operations. If c is negligible compared to t,

$l \simeq mt$

Analysing Control Structures
"For" loops: Nested Loops

- let's look at a nested loop that prints pairs:
- Step by Step Cost analysis:

```python
def print_pairs(n):
    for i in range(n):
        for j in range(n):
            print(i, j)    # O(1) operation
```

Part	Cost
Outer loop setup	$c_1 \times (n+1)$
Inner loop setup	$c_2 \times n \times (n+1)$
Print operation	$c_3 \times n^2$
Total	$c_1(n+1) + c_2n(n+1) + c_3n^2$

so the final complexity is:

$$\boxed{O(n^2)}$$

Analysing Control Structures
"For" loops: Nested Loops

- What if the inner loop depends on the outer loop?
- Step by Step Cost analysis:

```python
def triangular_loop(n):
    for i in range(n):
        for j in range(i + 1):
            print(i, j)     # O(1) operation
```

Line	Statement	Cost
1	`for i in range(n):`	$c_1 \times (n + 1)$
2	`for j in range(i + 1):`	$c_2 \times \sum_{i=0}^{n-1}(i+2)$
3	`print(i, j)`	$c_3 \times \sum_{i=0}^{n-1}(i+1)$

Why i+2 times?→

Suppose `i = 3` (so `j` should run from 0 to 3):

- `j = 0` → compare (0 ≤ 3)? ✅ → enter
- `j = 1` → compare (1 ≤ 3)? ✅ → enter
- `j = 2` → compare (2 ≤ 3)? ✅ → enter
- `j = 3` → compare (3 ≤ 3)? ✅ → enter
- `j = 4` → compare (4 ≤ 3)? ❌ → exit

Analysing Control Structures
"For" loops: Nested Loops

- What if the inner loop depends on the outer loop?
- Step by Step Cost analysis:

```
def triangular_loop(n):
    for i in range(n):
        for j in range(i + 1):
            print(i, j)    # O(1) operation
```

1. Simplifying $\sum_{i=0}^{n-1}(i+1)$:

$$\sum_{i=0}^{n-1}(i+1) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n-1}(i+1) = \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} 1 = \frac{n(n-1)}{2} + n$$

Simplify: Rewrite n as $\frac{2n}{2}$:

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

Factorize:

$$\frac{n^2+n}{2} = \frac{n(n+1)}{2}$$

Analysing Control Structures
"For" loops: Nested Loops

- What if the inner loop depends on the outer loop?
- Step by Step Cost analysis:

```python
def triangular_loop(n):
    for i in range(n):
        for j in range(i + 1):
            print(i, j)    # O(1) operation
```

2. Simplifying $\sum_{i=0}^{n-1}(i+2)$:

$$\sum_{i=0}^{n-1}(i+2) = \left(\sum_{i=0}^{n-1} i\right) + 2n$$

$$= \frac{(n-1)n}{2} + 2n$$

$$= \frac{n^2+n}{2}$$

Analysing Control Structures
"For" loops: Nested Loops

- What if the inner loop depends on the outer loop?
- Step by Step Cost analysis:

```python
def triangular_loop(n):
    for i in range(n):
        for j in range(i + 1):
            print(i, j)     # O(1) operation
```

Line	Statement	Cost
1	`for i in range(n):`	$c_1 \times (n + 1)$
2	`for j in range(i + 1):`	$c_2 \times \sum_{i=0}^{n-1}(i+2)$
3	`print(i, j)`	$c_3 \times \sum_{i=0}^{n-1}(i+1)$

Total Cost Expression:

$$c_1(n+1) + c_2\left(\frac{n^2+n}{2}\right) + c_3\left(\frac{n(n+1)}{2}\right)$$

so the final complexity is:

$$\boxed{O(n^2)}$$

Analysis: Nested Loops: A Harder Example

NESTED-LOOPS()

for $i \leftarrow 1$ **to** n
do
    for $j \leftarrow 1$ **to** $2i$
    do $k = j \ \ldots$
        while $(k \geq 0)$
        do $k = k - 1 \ \ldots$

Solution

- How do we analyze the running time of an algorithm that has complex nested loop?
- The answer write out the loops as summations and then solve the summations.
- To convert loops into summations, we work from inside-out.

```
NESTED-LOOPS()
for i ← 1 to n
do
    for j ← 1 to 2i
    do k = j ...
        while (k ≥ 0)
        do k = k − 1 ...
```

Analysis: A Harder Example

```
NESTED-LOOPS()
for i ← 1 to n
do for j ← 1 to 2i
    do k = j
        while (k ≥ 0)
        do k = k − 1
```

It is executed for k = j, j − 1, j − 2, . . . , 0. Time spent inside the while loop is constant. Let I() be the time spent in the while loop

$$I(j) = \sum_{k=0}^{j} 1 = j + 1$$

Analysis: A Harder Example

middle for loop.

```
NESTED-LOOPS()
for i ← 1 to n
do for j ← 1 to 2i  ◄
    do k = j
        while (k ≥ 0)
        do k = k − 1
```

Its running time is determined by i. Let M() be the time spent in the for loop:

$$
\begin{aligned}
M(i) &= \sum_{j=1}^{2i} I(j) \\
&= \sum_{j=1}^{2i} (j+1) \\
&= \sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1 \\
&= \frac{2i(2i+1)}{2} + 2i \\
&= 2i^2 + 3i
\end{aligned}
$$

Analysis: A Harder Example

Finally the *outer-most for* loop

```
NESTED-LOOPS()
for i ← 1 to n
do for j ← 1 to 2i
    do k = j
        while (k ≥ 0)
        do k = k − 1
```

Let T() be running time of the entire algorithm:

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} M(i) \\
&= \sum_{i=1}^{n} (2i^2 + 3i) \\
&= \sum_{i=1}^{n} 2i^2 + \sum_{i=1}^{n} 3i \\
&= 2\frac{2n^3 + 3n^2 + n}{6} + 3\frac{n(n+1)}{2} \\
&= \frac{4n^3 + 15n^2 + 11n}{6} \\
&= \Theta(n^3)
\end{aligned}
$$

Analysis: Class Activity

```
HARDER-NESTED-LOOPS(n)
for i = 1 to n
  do for j = 1 to i
    do k = j
      while (k > 0)
        do k = k - 1
```

Perform Cost wise analysis starting from inner loop

Analyzing Control Structures
Summery

- Algorithm usually proceeds from the inside out
- First determine the time required by individual instructions
- Second, combine the times according to the control structures that combine the instructions in the program
- Some control structures sequencing are easy to evaluate
- Others such as while loops are more difficult

Recursion

- Recursion provides an alternate of loops to solve a problem.
- Recursion is a function having a statement which call the same function.

Stack

- A stack is a last-in/first out memory structure. The first item referenced or removed from a stack is always the last item entered into the stack. For example,a pile of books.
- Memory for recursion calls is a Stack.

What's the structure of recursion?

- **Base cases-**One or more cases in which the function accomplished its task without the use of any recursive call.
- **Recursive cases-**One or more caes in which function accomplishes its task by using recursive calls to accomplish one or more smaller versions of task.

Think before using recursion

- What's the base case(s).
- How to divide the original problem into sub problems.
- How to merge the sub problem's results to get the final result.

- def factorial(n):

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Example: Fibonacci Sequence

The **Fibonacci sequence** is defined as:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \quad \text{for } n \geq 2$$

The series:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ \ldots$$

Analysing of For Loop for Computing

Function Fibiter(n)

{ Calculates the n-th term of the Fibonacci sequence}

```
i ← 1;
j ← 0
for k ← 1 to n do
    j ← i + j
    i ← j – i
return j
```

Fibonacci (5)

i = 1

j = 0

k =	**1**	**2**	**3**	**4**	**5**
j =	**1**	**1**	**2**	**3**	**5**
i =	**0**	**1**	**1**	**2**	**3**

- Time taken by the instructions inside the for loop is bounded above constant c
- Time taken by the for loop = nc
- Algorithm takes a time in O(n), Ω (n) and θ(n)
- Length of integer is important to determine the time taken by kth trip as the values of i & j are $f_{k-1}$and f_k respectively.

Recursive Call

- **function** *Fibrec(n)*

- **if** $n < 2$ **then return** n
- **else return** *Fibrec*(n - 1) + *Fibrec*(n - 2)

- Let $T(n)$ be the time taken by a call on *Fibrec*(n)

- If $n < 2$, the algorithm simply returns n, which takes some time constant time a

- Most of the work is spent in the two recursive calls which take time $T(n - 1)$ and $T(n - 2)$

- One addition involving f_{n-1} and f_{n-2} (values returned by the recursive calls)

Recursive Fibonacci (cont)

Let $h(n)$ be the work involved in the addition and control (we ignore the time spent inside the two recursive calls)

By definition of $T(n)$ and $h(n)$

$$T(n) = \begin{cases} a & \text{if } n = 0 \text{ or } n = 1 \\ T(n - 1) + T(n + 2) + h(n) & \text{otherwise} \end{cases}$$

Solving this recurrence gives us:

$$T(n) = O(2^n)$$

- **Worst-case time**: $O(2^n)$
- **Best-case time**: $\Theta(1) \to$ Only for `n = 0 or 1`
- **Average-case**: Still close to **$O(2^n)$**, since most `n > 1` follow the same recursive explosion.

Recursive Fibonacci (cont)

```
                       fib(5)
                   /           \
             fib(4)             fib(3)
            /     \            /     \
       fib(3)   fib(2)   fib(2)   fib(1)
       /   \     /   \    /   \
  fib(2) fib(1) fib(1) fib(0) fib(1)
  /   \
fib(1) fib(0)
```

Recursive Fibonacci: Memoization (cont)

Memoization is a technique used in programming to **speed up recursive functions** by **storing (caching)** the results of expensive function calls and **reusing** them when the same inputs occur again.

```
memo = {}

def fib(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```

- **Time Complexity**: O(n)
- Each subproblem is solved **only once** and stored.

Memo updates (in sequence):

- `memo[2] = 1`
- `memo[3] = 2`
- `memo[4] = 3`
- `memo[5] = 5`

Comparison: Fibonacci using Loop, recursive function and Recursive with Memoization

Method	Time Complexity	Space Complexity	Calls	Practical Use
Iterative	`O(n)`	`O(1)`	None	✅ Best
Recursive	`O(2^n)`	`O(n)`	Exponential	❌ Not for large n
Memoized	`O(n)`	`O(n)`	Linear	✅ Good

While and Repeat Loops

- Usually harder to analyze than for loops - there is no a priori way to determine the amount of iterations through the loop

- Need to better understand how the value of the function decreases

While and Repeat Loops

- It is important to find a function of the variables involved in controlling the while / repeat loop.
- Binary search illustrates the analysis of while loops.
- Purpose is to find x in array T[1. . n] which appears in T at least once.

Binary Search: While Loop

```
def binary_search_iter(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Operations:

- Initialize `low` and `high`.
- Execute a `while` loop until `low <= high`.
- Inside the loop:
    - Compute `mid`.
    - Compare `arr[mid]` with `target`.
    - Return `mid` if found, or update `low` or `high`.
- Return `-1` if the target is not found.

Binary Search: While Loop

The key to the time complexity lies in the number of `while` loop iterations. Binary search divides the search space in half each iteration:

- **Initial search space**: $high - low + 1 = (n-1) - 0 + 1 = n.$
- **After 1st iteration**:
  - If `arr[mid] < target`, set `low = mid + 1`, reducing the search space to roughly $n/2$.
  - If `arr[mid] > target`, set `high = mid - 1`, reducing the search space similarly.
- **After 2nd iteration**: Search space is $n/4$.
- **After k-th iteration**: Search space is $n/2^k$.

The loop continues until:

- The target is found (return inside loop), or
- $low > high$, which occurs when the search space is empty.

The search space becomes size 1 (or less) when:

$$n/2^k \leq 1$$

$$2^k \geq n$$

$$k \geq \log_2 n$$

Worst-case iterations: Occurs when the target is not in the array, and the loop runs until the search space is empty. This takes $O(\log n)$ iterations.

Binary Search: While Loop

Operation	Cost per Execution	Frequency	Total Cost
Initialize `low`, `high`	$O(1)$	1	$O(1)$
While condition check	$O(1)$	$O(\log n)$	$O(\log n)$
Loop body (compute `mid`, comparisons, updates)	$O(1)$	$O(\log n)$	$O(\log n)$
Return `-1`	$O(1)$	At most 1	$O(1)$

Sum the total costs:

- Initialization: $O(1)$.
- While loop (condition + body): $O(\log n) \times O(1) = O(\log n)$.
- Final return: $O(1)$.

Total:

$$O(1) + O(\log n) + O(1) = O(\log n)$$

Binary search: Recursive

```
def binary_search(arr, target, low, high):
    if low > high:
        return -1
    mid = (low + high) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] > target:
        return binary_search(arr, target, low, mid-1)
    else:
        return binary_search(arr, target, mid+1, high)
```

Binary search: Recursive

To find the total time complexity, we model the algorithm's runtime using a recurrence relation. Let $T(n)$ represent the time to search an array of size n, where $n = high - low + 1$ is the size of the current search range.

- **Base Case**: If `low > high`, the function returns `-1`.
  - Cost: $O(1)$.
  - So, $T(1) = O(1)$ (or for an empty range, $T(0) = O(1)$).
- **Recursive Case**:
  - Non-recursive work per call: Computing `mid`, comparisons, and preparing arguments for the recursive call are all $O(1)$.
  - Recursive call: The algorithm makes one recursive call on a subproblem of size at most $n/2$:
    - If `arr[mid] < target`, call `binary_search_rec(arr, target, mid + 1, high)`.
      - New range: $high - (mid + 1) + 1 = high - mid$.
      - Since `mid = (low + high) // 2`, the new size is roughly $n/2$.
    - If `arr[mid] > target`, call `binary_search_rec(arr, target, low, mid - 1)`.
      - New range: $(mid - 1) - low + 1 = mid - low$.
      - Similarly, size is roughly $n/2$.

Binary search: Recursive

- **Recurrence**:

$$T(n) = T(n/2) + O(1)$$

The $O(1)$ term accounts for all non-recursive operations (comparisons, arithmetic, etc.).

Unroll the recurrence:

$$T(n) = T(n/2) + 1$$

$$T(n/2) = T(n/4) + 1$$

$$T(n) = [T(n/4) + 1] + 1 = T(n/4) + 2$$

$$T(n/4) = T(n/8) + 1$$

$$T(n) = T(n/8) + 3$$

After k iterations:

$$T(n) = T(n/2^k) + k$$

Binary search: Recursive

- **Base case**: Assume the base case occurs when the size is 1 (i.e., $n/2^k = 1$).

  $$n/2^k = 1 \implies 2^k = n \implies k = \log_2 n$$

  Assume $T(1) = 1$ (constant time for a single element).

  $$T(n) = T(1) + \log_2 n = 1 + \log_2 n$$

- **Time complexity**: $O(\log n)$.

Recurrence Relation

- A **recurrence relation** is a mathematical equation that defines a sequence or function in terms of its values at smaller inputs.
- In the context of **algorithm analysis**, it's used to describe the time or space complexity of a recursive algorithm by expressing the cost of solving a problem of size n in terms of the cost of solving smaller subproblems, plus any additional work done outside the recursive calls.

A recurrence relation for an algorithm's time complexity $T(n)$ typically looks like:

$$T(n) = (\text{cost of recursive calls}) + (\text{cost of non-recursive work})$$

With a **base case** that defines $T(n)$ for small inputs (e.g., $n = 1$ or $n = 0$).

Recurrence Relation (cont'd)

Key Components

1. **Recursive Part**
   - Specifies $T(n)$ in terms of earlier values (e.g., $T(n-1)$, $T(n-2)$, etc.).
   - Example: $T(n) = T(n-1) + T(n-2)$ (Fibonacci-like).
2. **Base Case(s)**
   - Explicit values for small n so the recursion terminates.
   - Example: $T(0) = 1,\ T(1) = 1$.

Recurrence Relation (cont'd)

Example 1: Fibonacci Sequence

- **Relation:**

$$T(n) \;=\; T(n-1) + T(n-2), \quad n \geq 2$$

- **Base Cases:**

$$T(0) = 0, \quad T(1) = 1$$

This recurrence fully determines every Fibonacci number.

Example 2: Merge Sort Time Complexity

When analyzing Merge Sort, the running time $T(n)$ satisfies:

$$T(n) \;=\; 2\,T\left(\tfrac{n}{2}\right) \;+\; n, \quad T(1) = O(1).$$

- Here, $T(n)$ depends on two subproblems of size $n/2$ plus $\Theta(n)$ for the merging step.

Recurrence Relation (cont'd)

Uses in Algorithm Analysis

- **Divide-and-Conquer Algorithms:** Many divide-and-conquer runtimes are modeled by recurrences of the form

$$T(n) = a\,T\left(\tfrac{n}{b}\right) + f(n).$$

  Solving these recurrences (via Master Theorem, recursion trees, or substitution) yields the algorithm's asymptotic complexity.

Why Use Recurrence Relations?

- **Model Recursive Algorithms**: They capture how recursive algorithms break down problems.
- **Analyze Complexity**: Solving the recurrence gives the algorithm's time complexity.
- **Optimize Design**: Understanding the recurrence helps identify bottlenecks and improve algorithms.

Recurrence Relation (cont'd)

Common methods include:

1. **Iteration (Unrolling)**: Expand the recurrence to find a pattern.
2. **Recursion Tree**: Visualize the work at each level of recursion.
3. **Master Theorem**: For recurrences like $T(n) = aT(n/b) + f(n)$.
4. **Substitution Method**: Guess a solution and prove it using induction.