Algorithm Analysis

CMPT 145

Cost of Computation

- Programs use time, memory, other valuable resources
- Resources are not free! They have costs.
- We need a way to assess these costs scientifically (quantitatively).

Algorithm vs Program

- Algorithm
  - A sequence of actions that describe how to perform a task or solve a problem.
  - Makes no commitment to a programming language.
- Program
  - A concrete realization of an algorithm, expressed in a programming language.
  - An algorithm can be realized by many different programs.
- When we assess the costs of a program, we ignore the variable aspects, and assess the underlying algorithm.
- We will use abstraction to ignore variable aspects.

Abstraction #1: Time

- Consider two scenarios:
  - Program A executed on a computer built in 2017.
  - Program A executed on a computer built in 1967.
- **Question:** which program runs faster?
- **Answer:** We have to separate the costs of the program from the abilities of the computer.
- **Abstraction:**
  - Measure time in terms of computational steps.
  - Program A requires a number of steps.
  - Different computers do the steps faster or slower.

What is a computational step?

- Any of the following is a computational step:
  - An arithmetic, logical, or relational operation
  - Assigning a value to a variable
  - Accessing an element in a Python list (or numpy array)
  - Calling a function, returning a value
- Cost of expressions used as operands and function bodies must be assessed separately
- We will review this idea later.

How many computational steps?

(a) Example 1:

```
x = 3*y+4
```

(b) Example 2:

```
x = data[2]*data[3]
```

(c) Example 3:

```
x = 3*math.sqrt(data[i+1])
```

How many computational steps?

(a) Example 4:

```
total = 0
for i in range(10):
    total = total + 1
```

(b) Example 5:

```
total = 0
for i in range(N):
    total = total + 1
```

(c) Example 6:

```
total = 0
for i in range(N):
     for j in range(N):
         total = total + 1
```

Abstraction #2: Input size

- Consider two scenarios:
  - Program A executed on a list of size 10.
  - Program A executed on a list of size 1000.
- **Question:** which program runs faster?
- **Answer:** We have to express the cost as a function of the input size.
- **Abstraction:**
  - Parameterize the size of the problem.
  - Express the number of steps taken as a function of the problem size.
  - Program A may take more steps on larger lists.

Parameterization

- Parameterization assigns a mathematical name to a quantity that could change, or is unknown.
    - e.g. A list of size N, a box of size $L \cdot W \cdot H$, etc.
- Express the size of a problem using parameters
    - Usually expressed as N, M, etc.
- Express the number of steps required by an algorithm in terms of the problem size
    - e.g. Algorithm A takes $3N + 12$ steps to complete for input of size N

Parameterization examples (simple)

What are the input size parameters for the following examples?

(a) Function to find the maximum value in a list (1D).

(b) Function to calculate the average value of a list (1D).

(c) Function to display all permutations of a list of items.

Parameterization examples (more simple)

What are the input size parameters for the following examples?

(a) Function to check if a square is magic (e.g., Assignment 1).

(b) Function to evaluate an expression (Assignment 2), if:
- The symbols are already in a list, e.g, : `['(', 3, '+', 1, ')']`
- The expression is a string, e.g, : `'( 3 + 1 )'`

(c) Function to enqueue a value to a given queue (Assignment 2).

(d) Function to remove a given value from a node-based list (Assignment 3).

Parameterization examples

The Sieve of Eratosthenes

- Uses a list of boolean values
  - One entry per positive integer from 2 to n
  - Value at index i reflects whether i is prime or not
  - Initially, all `True`
- For every number i from $2 \dots n$,
  - If i is prime, mark its multiples as not prime
  - e.g. 2 is a prime number, so mark 4, 6, 8, etc. as not prime

What are the parameter(s) that indicate problem size?

Parameterization examples (trickier)

The Gambler's Ruin Problem

- 50%-50% chance to win
- Winning earns 1 dollar, losing removes 1 dollar
- Gambler has a stake and a goal
- Gambler plays games until success or failure.
- To estimate the probability, we simulate the game a number of times.

What are the parameters that indicate problem size?

Abstraction #3: Categories

- Consider two scenarios:
  - Program A requires $3N + 12$ steps.
  - Program B requires $3N + 11$ steps.
  - (Both programs solve the same problem.)
- **Question:** which program runs faster?
- **Answer:** We ignore differences that don't matter.
- **Abstraction:**
  - Express the cost of a program by placing it in a category.
  - All programs in the category are considered equally efficient.
  - Program A and Program B are in the same category.

How much difference makes a difference?

- Suppose Program A requires $3N + 12$ steps.
- For very large N, the extra 12 steps is negligible.
- Our categories ignore negligible details.
- For very large N, costs are (nearly) proportional to N.
- Our categories ignore constants of proportionality.

Asymptotic Analysis: Large problem sizes

- When we consider very large problem sizes, we ignore
  - Negligible costs
  - Constants of proportionality
- This technique is called Asymptotic analysis
- We use Big-O Notation to indicate the results of asymptotic analysis.
  - Suppose Program A requires $3N + 12$ steps.
  - We say Program A asymptotically requires $O(N)$ steps.

Big-O Notation

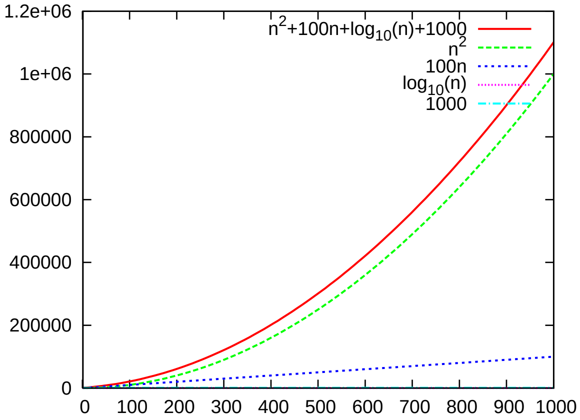
- Big-O is a formal notation for expressing an algorithm's asymptotic behaviour
  - The O indicates asymptotic analysis.
  - e.g. $O(N)$, $O(N^3)$, $O(N^k)$ etc.
  - The N is a problem size parameter, for a given algorithm.
- $O(N^2)$ means that computational costs increase no more quickly than N^2.

Big-O Categories

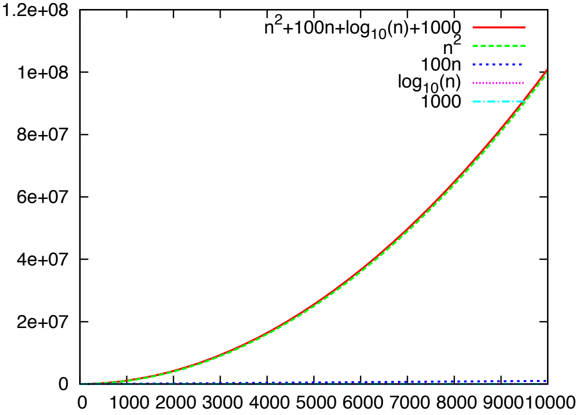
$$
\begin{aligned}
O(1) \quad &< \quad O(\log_2 N) < O(N) < O(N \log_2 N) \\
&< \quad O(N^2) < O(N^3) < \ldots < O(N^k) \\
&< \quad O(2^N) < O(N!) < O(N^N)
\end{aligned}
$$

- Here, $<$ means increases more slowly than, in the context of increasing N.
- The listing here is not complete. For example, $O(N^2 \log N)$ could be included.

Big-O categories, visually



Big-O categories, bigger values of n



What Big-O Categories really mean

- Informally, $O(N^2)$ means that computational costs increase no more quickly than N^2.
- But N also increases no more quickly than N^2
- A Big-O category suggests an upper limit on the way costs increase with N
- When we categorize an algorithm or program, we want the upper limit to be
    - As far “left” in the $<$ order as possible.
    - As “low” as possible in a graph.

Abstraction #4: Worst-case behaviour

- Consider linear search for a given value in a given list.
  - The value appears as the first value in the list.
  - The value appears as the last value in the list.
  - The value does not appear in the list.
- **Question:** Which of the cases is representative of the cost of a linear search?
- **Answer:** We use the worst-case, pessimistically.
- **Abstraction:**
  - The worst-case behaviour is used to categorize an algorithm.

Clarifying worst-case behaviour

- The worst-case for an algorithm forces it to do the maximum amount of work for a given problem size.
- **It is always wrong to mention problem size when discussing best and worst case behaviour.**
- The best case is not when a problem is small.
- The worst case is not when a problem is big.
- Many algorithms show no difference between best and worst cases.
- We would prefer to use average-case in principle, but except for really trivial algorithms, average-case analysis is usually too hard.

Examples: Worst-case behaviour

In terms of the time costs, what are the best and worst cases for the following?

(a) Linear search in a list.

(b) Binary search in a list.

(c) Calculating the average of a list of values.

(d) Inserting a value into a node-based chain of nodes.

(e) Gambler's ruin.

(f) Sieve of Eratosthenes.

Exercise 1

Categorize the following program in terms of Big-O, by counting steps. What is the Big-O analysis for the best and worst case?

```python
def roll_dice():
    return random.randint(1,6) + random.randint(1,6)

def rolling(n_rolls):
    low_rolls = 0     # number rolls whose sum is <= 4
    high_rolls = 0    # number rolls whose sum is >= 10
    dice_rolls = [0]*n_rolls
    for roll in range(n_rolls):
        dice_rolls[roll] = roll_dice()
        if dice_rolls[roll] <= 4:
            low_rolls += 1
        elif dice_rolls[roll] >= 10:
            high_rolls += 1
    print(low_rolls,high_rolls)
    print(dice_rolls)
```

Asymptotic Analysis: A simplified algebra

- Before we perfect the counting of steps, we introduce a few rules about Big-O notation.
- This will make counting steps so much easier.
- None of this is hard, but it does look like math.
- We start with rules and simplifications.
- Practical examples will come after.

Asymptotic Analysis Principles

Identities

Let A, B be any mathematical expression. In asymptotic analysis, the following identities hold:

$$\begin{aligned} O(A) + O(B) &= O(A + B) \\ O(A) \times O(B) &= O(A \times B) \\ A \times O(B) &= O(A \times B) \end{aligned}$$

Asymptotic Analysis Principles

Replacing constants with 1

In asymptotic analysis:

- If A is a constant, $O(A) = O(1)$.
- If A is a constant, and B is any expression, $O(A \times B) = O(B)$.

Drop lower order terms

In asymptotic analysis, if $O(A) < O(B)$, then $O(B) + O(A) = O(B)$.

Example

Use the rules for Big-O to simplify this expression:

$$
\begin{aligned}
& O(3N + N \log N + 4) \\
& = O(3N) + O(N \log N) + O(4) \\
& = O(N) + O(N \log N) + O(1) \\
& = O(N \log N)
\end{aligned}
$$

Exercise 2

Use the rules for Big-O to simplify these expressions:

(a) $O(6N^2 + 2N \log N + 3)$

(b) $O(6 \times 2^N + 2N^2 \log N + 10^{1700} \times N)$

Using the simplified algebra: expressions and assignments

- For best and worst case analysis, don't count steps.
- Use Big-O as an abstraction for counting.
- Annotate each line of code with a $O(\cdot)$ expression.
- Use the identities and simplifications.

Exercise 3

What's the Big-O for each line of code?

```
x = 3*y+4

y = data[2]*data[3]

z = 3*abs(data[i+1])

alist = [0]*N
```

Using the simplified algebra: single loop

- Analyze the body of the loop using Big-O
- Determine how often the loop is repeated.
- Multiply.

Method 1: Step analysis

Example 1:

```
1 total = 0
2 for i in range(10):
3     total = total + 1
```

- Line 1: 1 step
- Just line 3, once: 3 steps
- Just line 2, over-all: 10 assignments, 9 additions, so 19 steps
- Line 3 is repeated 10 times, so 30 steps
- Totals: $1 + 19 + 30 = 50$ steps
- Asymptotically, 50 is $O(1)$
- Literally, the problem size here is not a variable.

Method 2: Simplified Steps

Example 1:

```
1 total = 0
2 for i in range(10):
3     total = total + 1
```

- Line 1: 1 step, so $O(1)$
- Just line 3, once: 3 steps, so $O(1)$
- Just line 2, over-all: 10 assignments, 9 additions, 19 steps, so $O(1)$
- Line 3 is repeated 10 times, so $10 \times O(1) = O(1)$
- Totals: $O(1) + O(1) + O(1) = O(1 + 1 + 1) = O(1)$
- Literally, the problem size here is not a variable.

Method 1: Step analysis

Example 2:

```
1 total = 0
2 for i in range(N):
3     total = total + 1
```

- Line 1: 1 step
- Just line 3, once: 3 steps
- Just line 2, over-all: N assignments, $N - 1$ additions, so $2N - 1$ steps.
- Line 3 is repeated N times, so $3N$ steps
- Detailed step analysis: $1 + 2N - 1 + 3N = 5N$
- Asymptotic category: $5N$ is $O(N)$.

Method 2: Simplified steps

Example 2:

```
1 total = 0
2 for i in range(N):
3     total = total + 1
```

- Line 1: 1 step, so $O(1)$
- Just line 3, once: 3 steps, so $O(1)$
- Just line 2, over-all: N assignments, $N - 1$ additions, $2N - 1$ steps, so $O(N)$
- Line 3 is repeated N times, so $N \times O(1) = O(N)$
- Totals: $O(1) + O(N) + O(N) = O(2N + 1) = O(N)$

Exercise 4

```
1 total = 0
2 for i in range(N):
3     total = total + 1
```

- How many steps would Line 2 have to do so that the whole script is worse than $O(N)$?

Exercise 5

Analyze the following code, given each of the assumptions below:

```
total = 0
for i in range(N):
    total = some_function(i,N)
```

(a) If `some_function(i,N)` were $O(1)$?

(b) If `some_function(i,N)` were $O(\log N)$?

(c) If `some_function(i,N)` were $O(N^2)$?

Variations in terminology

All of the following mean the same thing:

- Analyze the program for its asymptotic run time costs.
- What is the asymptotic time complexity of the program?
- Use Big-O to categorize the run time of the program.
- Express the run time costs of the program using Big-O notation.

Exercise 6

Express the run time costs of the following program using Big-O notation:

```
total = 0
i = 0
while (i < N):
    total = total + 1
    i = i + 1
```

Logarithmic loops

```
total = 0
i = 1
while (i < N):
    total = total + 1
    i = i * 2
```

Question: How many times does the loop-body execute?
Insight: The values for i are: 1, 2, 4, 8, …
Answer: We can double i at most $\log_2 N$ times before $i > N$

Logarithmic loops

```
total = 0
i = N
while i > 1:
    total = total + 1
    i = i // 2
```

Question: How many times does the loop-body execute?
Insight: The values for i are: $N, N/2, N/4, \ldots$
Answer: We can half $i = N$ at most $\log_2 N$ times before $i \leq 1$.

Using the simplified algebra: sequential loops

- Analyze each loop separately
- Add.

Exercise 7

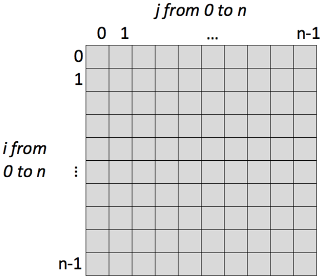
```
total = 0
for i in range(N):
    total = total + 1

for j in range(N):
    total = total + 1
```

Nested loops with independent limits

- Start with the inner-most loop
- Analyze to find the number of steps
- Abstraction: treat the inner-most loop as a statement with the analyzed cost.
- Work outward, one loop at a time.

Exercise 8

```
total = 0
for i in range(N):
    for j in range(N):
        total = total + 1
```



Exercise 9

```python
total = 0
for i in range(N):
    for j in range(M):
        total = total + 1
```

Assume M and N are separate input size parameters.

Nested loops with dependent limits

```
1 total = 0
2 for i in range(N):
3     for j in range(i):
4         total = total + 1
```

- When `i == 0`, lines 3-4 repeated 0 times
- When `i == 1`, lines 3-4 repeated 1 times
- When `i == 2`, lines 3-4 repeated 2 times
- When `i == N-1`, lines 3-4 repeated N-1 times
- In general, lines 3-4 are repeated $O(i)$ times

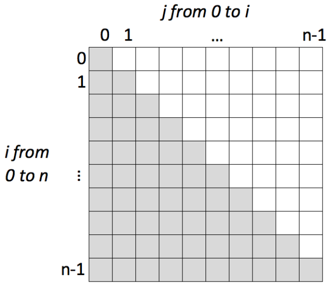
Nested loops with dependent limits

```
1 total = 0
2 for i in range(N):
3       for j in range(i):
4           total = total + 1
```

- In general, lines 3-4 are repeated $O(i)$ times
- Total cost: sum the costs for `i` from 0 to $N-1$
- Total: $O(0) + O(1) + O(2) + \cdots + O(i) + \cdots + O(N-1)$

An identity you must memorize

$$\sum_{i=1}^{n} i \equiv 1 + 2 + 3 + 4 + \cdots + n = \frac{n(n+1)}{2}$$



Nested loops with dependent limits

```
1 total = 0
2 for i in range(N):
3       for j in range(i):
4           total = total + 1
```

- In general, lines 3-4 are repeated $O(i)$ times
- Total cost: sum the costs for `i` from 0 to $N-1$
- Total: $O(0) + O(1) + O(2) + \cdots + O(N-1) = O(N^2)$

Nested loops with dependent limits

- Start with the inner-most loop
- Analyze to find the number of steps expressed using dependency
- Abstraction: treat the inner-most loop as a statement with the analyzed cost.
- Sum the total costs of the dependent loop
- Work outward, one loop at a time.

If statements

- Analyze the cost of the condition(s)
- Analyze the cost of each branch
- Worst case analysis: Use the cost of the most expensive branch only
- Best case analysis: Use the cost of the least expensive branch only

Exercise 10

What is the asymptotic time complexity of the following program?

```
sign = 0
if N < 0:
    sign = -1
else:
    sign = 1
```

Exercise 11

Analyze the following program for its asymptotic run time costs.

```
total = 0
i = 0
while i < N:
    if i % 2 == 0:
        total = total + 1
    i = i + 1
```

Exercise 12

Analyze the following program for its asymptotic run time costs.

```
total = 0
i = 0
while i < N:
    if i < N//2:
        total = total + i
    else:
        total = total + 2*i
    i = i + 1
```

Exercise 13

Analyze the following program for its asymptotic run time costs.

```
total = 0
i = 0
while i < N:
    if i % 2 == 0:
        for j in range(N):
            total = total + j
    else:
        total = total + 2*i
    i = i + 1
```

Exercise 14

What is the asymptotic time complexity of the following program?

```
total = 0
for value in list_of_numbers:
    if value % 2 == 0:
        total = total + 1
```

Exercise 15

What is the asymptotic time complexity of the following program?

```
total = 0
for value in list_of_numbers:
    if value < k:
        total = total + value
    else:
        total = total + 2*value
```

Exercise 16

```
total = 0
for value in list_of_numbers:
    if value % 2 == 0:
        total = total + 1
    else:
        for i in range(len(list_of_numbers)):
            total = total + 1
```

Exercise 17

Categorize the following program in terms of Big-O, by counting steps. What is the Big-O analysis for the best and worst case?

```
def roll_dice():
    return random.randint(1,6) + random.randint(1,6)

def rolling(n_rolls):
    low_rolls = 0     # number rolls whose sum is <= 4
    high_rolls = 0    # number rolls whose sum is >= 10
    dice_rolls = [0]*n_rolls
    for roll in range(n_rolls):
        dice_rolls[roll] = roll_dice()
        if dice_rolls[roll] <= 4:
            low_rolls += 1
        elif dice_rolls[roll] >= 10:
            high_rolls += 1
    print(low_rolls,high_rolls)
    print(dice_rolls)
```

Exercise 18

```python
# n_players is a pre-defined integer
round_robin = np.zeros([n_players,n_players],dtype=int)
games_per_match = 5
for p1 in range(n_players):
    for p2 in range(p1+1,n_players):
        p1_games_won = random.randint(1,games_per_match)
        round_robin[p1,p2] = p1_games_won
        round_robin[p2,p1] = games_per_match - p1_games_won
print(round_robin)
```