Big-O

• Given function $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ so that
  $$f(n) \leq cg(n) \text{ for } n \geq n_0$$

• Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $10 \leq n(c - 2)$
  - $n \geq 10/(c-2)$
  - Pick $c = 3$ and $n_0 = 10$
Big-O

• 4n - 2 is O(n)
  - Need a c > 0 and n₀ ≥ 1 so that
    \[4n - 2 \leq cn \text{ for } n \geq n₀\]
  - true for c = 4 and n₀ = 1

• 5n³ + 10n² + 4n + 2 is O(n³)
  - Need a c > 0 and n₀ ≥ 1 so that
    \[5n³ + 10n² + 4n + 2 \leq cn³ \text{ for } n \geq n₀\]
  - true for c = 21 and n₀ = 1

• 2 \log₂ n + 3 is O(\log₂ n)
  - Need a c > 0 and n₀ ≥ 1 so that
    \[2\log₂ n + 3 \leq c \log₂ n \text{ for } n \geq n₀\]
  - true for c = 5 and n₀ = 2

Big-O

• Given function f(n) and g(n), we say that \textbf{f(n) is O(g(n))} if there are positive constants c and n₀ so that \(f(n) \leq cg(n)\) for \(n \geq n₀\)

<table>
<thead>
<tr>
<th></th>
<th>f(n) is O(g(n))</th>
<th>g(n) is O(f(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>g(n) grows more</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>f(n) grows more</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>g(n) and f(n) has same growth</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Ex 1

define a type Int_Array as an array of integers with a specific range.

procedure Measure (A : Int_Array ) is
  Sum : Integer := 0;
begin
  for I in A’range loop
    for J in A’range loop
      Sum := Sum + A(J);
    end loop;
  end loop;
end Measure;

<table>
<thead>
<tr>
<th>Statement</th>
<th>Runs in X time</th>
<th>Executes # of times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Sum is initialized</td>
<td>Constant1</td>
<td>1</td>
</tr>
<tr>
<td>Array of size n is created</td>
<td>Constant2</td>
<td>1</td>
</tr>
<tr>
<td>Variable I is created and initialized</td>
<td>Constant3</td>
<td>1</td>
</tr>
<tr>
<td>I is tested against A’range (n)</td>
<td>Constant4</td>
<td>n+1</td>
</tr>
<tr>
<td>Variable J is created and initialized</td>
<td>Constant5</td>
<td>n</td>
</tr>
<tr>
<td>J is tested against A’range (n)</td>
<td>Constant6</td>
<td>n(n+1)</td>
</tr>
<tr>
<td>Sum is incremented by A(J)</td>
<td>Constant7</td>
<td>n^2</td>
</tr>
<tr>
<td>J is incremented by 1</td>
<td>Constant8</td>
<td>n^2</td>
</tr>
<tr>
<td>I is incremented by 1</td>
<td>Constant9</td>
<td>n</td>
</tr>
</tbody>
</table>
type Int_Array is array (Integer range <>) of Integer;

procedure Measure (A : Int_Array) is
  Sum : Integer := 0;
begin
  for I in A'range loop
    for J in 1 .. I loop -- only change to Ex 1
      Sum := Sum + A(J);
    end loop;
  end loop;
end Measure;

CQ – Ex 2

<table>
<thead>
<tr>
<th>Variable J is created and initialized</th>
<th>Constant5</th>
</tr>
</thead>
<tbody>
<tr>
<td>J is tested against I</td>
<td>Constant6</td>
</tr>
<tr>
<td>Sum is incremented by A(J)</td>
<td>Constant7</td>
</tr>
<tr>
<td>J is incremented by 1</td>
<td>Constant8</td>
</tr>
</tbody>
</table>

1. \( N, \quad N(N+1), \quad N^2, \quad N \)
2. \( N, \quad N(I+1), \quad N^2, \quad N^2 \)
3. \( N, \quad N(I+1), \quad N^2, \quad N^2 \)
4. I still don’t get it
Ex 3

type Int_Array is array (Integer range <>) of Integer;

procedure Measure (A : Int_Array ) is
  Sum : Integer := 0;
begin
  for I in A'range loop
    for J in 1 .. 4 loop -- only change to Ex 2
      Sum := Sum + A(I); -- only change to Ex 2
    end loop;
  end loop;
end Measure;

BigO3.adb

CQ – Ex 3

<table>
<thead>
<tr>
<th>Variable J is created and initialized</th>
<th>Constant5</th>
</tr>
</thead>
<tbody>
<tr>
<td>J is tested against I</td>
<td>Constant6</td>
</tr>
<tr>
<td>Sum is incremented by A(J)</td>
<td>Constant7</td>
</tr>
<tr>
<td>J is incremented by 1</td>
<td>Constant8</td>
</tr>
</tbody>
</table>

1. $N, \quad N*(I+1), \quad N*I, \quad N*I$

2. $N, \quad N*5, \quad N*4, \quad N*4$

3. $N, \quad N*5, \quad 4, \quad 4$

4. I still don’t get it
function Factorial (N : in Natural ) return Positive is begin
  if N = 0 then
    return 1;
  else
    return N * Factorial (N-1);
  end if;
end Factorial;

CQ – Ex 4

How long time does executing the Factorial algorithm take?

1. O(n)
2. O(n²)
3. log n
4. 42
Divide and Conquer

- It is an algorithmic design paradigm that contains the following steps
  - **Divide**: Break the problem into smaller sub-problems
  - **Recur**: Solve each of the sub-problems recursively
  - **Conquer**: Combine the solutions of each of the sub-problems to form the solution of the problem

Represent the solution using a recurrence equation

Merge Sort

- **Divide**: Split the array into into two subarrays $A(p .. mid)$ and $A(mid+1 .. r)$, where $mid$ is $(p + r)/2$

- **Conquer** by recursively sorting the two subarrays $A(p .. mid)$ and $A(mid+1 .. r)$

- **Combine** by merging the two sorted subarrays $A(p .. mid)$ and $A(mid+1 .. r)$ to produce a single sorted subarray $A(p .. r)$
Merge

• **Input:** Array $A$ and indices $p$, $mid$, $r$ such that
  - $p \leq mid < r$
  - subarray $A(p .. mid)$ is sorted and subarray $A(mid+1 .. r)$ is sorted

• **Output:** single sorted array $A(p .. r)$

• $T(n) = O(n)$, where $n=r-p+1 = \# \text{ of elements being merged}$

Merge Sort Analysis

• **The base case:** when $n = 1$, $T(n)=O(1)$

• When $n \geq 2$, time for merge sort steps:
  - **Divide:** Compute mid as the average of $p$, $r$
    ⇒ cost = $O(1)$
  - **Conquer:** Solve 2 subproblems, each of size $n/2$
    ⇒ cost = $2T(n/2)$
  - **Combine:** merge to an $n$ element subarray
    ⇒ cost = $O(n)$

\[
T(n) = \begin{cases} 
  O(1) & \text{if } n = 1 \\
  2T(n/2) + O(n) + O(1) & \text{if } n > 1 
\end{cases}
\]
Solving Recurrences: Iteration

\[ T(n) = \begin{cases} 
  c & n = 1 \\
  aT\left(\frac{n}{b}\right) + cn & n > 1 
\end{cases} \]

- \( T(n) = \)
  \( aT(n/b) + cn \)
  \( a(aT(n/b/b) + cn/b) + cn \)
  \( a^2T(n/b^2) + cna/b + cn \)
  \( a^2T(n/b^2) + cn(a/b + 1) \)
  \( a^2(aT(n/b^2/b) + cn/b^2) + cn(a/b + 1) \)
  \( a^3T(n/b^3) + cna^2/b^2 + cn(a/b + 1) \)
  \( a^3T(n/b^3) + cn(a^2/b^2 + a/b + 1) \)
  \[ \vdots \]
  \( a^kT(n/b^k) + cn(a^{k-1}/b^{k-1} + a^{k-2}/b^{k-2} + \ldots + a^2/b^2 + a/b + 1) \)
\[ T(n) = \begin{cases} \frac{c}{b} & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases} \]

- So we have
  - \( T(n) = a^n T(n/b^k) + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1) \)

- For \( k = \log_b n \)
  - \( n = b^k \)
    - \( T(n) = a^n T(1) + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1) \)
      - \( = a^k c + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1) \)
      - \( = c a^k + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1) \)
      - \( = cn a^k/b^k + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1) \)
      - \( = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1) \)

- What if \( a = b? \)
  - \( T(n) = cn(k + 1) \)
    - \( = cn(\log_b n + 1) \)
    - \( = O(n \log n) \)

\[
\begin{align*}
T(n) &= O(1) & n = 1 \\
2T(n/2) + O(n) + O(1) & n > 1
\end{align*}
\]
The Master Method

• Given: a *divide and conquer* algorithm
  - An algorithm that divides the problem of size *n* into *a* subproblems, each of size *n/b*

  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function *f(n)*

  - The *master method* provides a simple “*cookbook*” solution

Simplified Master Method

• \[ T(n) = aT(n/b) + cn^k, \]
  where \(a, c > 0\) and \(b > 1\)

\[
T(n) = \begin{cases} 
  O\left(\frac{n}{\log_b a}\right) & a \geq b^k \\
  O\left(\frac{n^k}{\log_b n}\right) & a = b^k \\
  O\left(n^k\right) & a \leq b^k 
\end{cases}
\]
The Towers of Hanoi

- **Goal**: Move stack of rings to another peg
  - May only move 1 ring at a time
  - May never have larger ring on top of smaller ring

For simplicity, suppose there were just 3 disks

Since we can only move one disk at a time, we move the top disk from A to B.
The Towers of Hanoi

For simplicity, suppose there were just 3 disks

We then move the top disk from A to C.

The Towers of Hanoi

For simplicity, suppose there were just 3 disks

We then move the top disk from B to C.
The Towers of Hanoi

For simplicity, suppose there were just 3 disks

We then move the top disk from A to B.

We then move the top disk from C to A.
The Towers of Hanoi

For simplicity, suppose there were just 3 disks

We then move the top disk from C to B.

We then move the top disk from A to B.
The Towers of Hanoi

For simplicity, suppose there were just 3 disks

\[ A \quad B \quad C \]

and we’re done!

The problem gets more difficult as the number of disks increases...

---

The Towers of Hanoi

- 1 ring \( \rightarrow \) 1 operation
- 2 rings \( \rightarrow \) 3 operations
- 3 rings \( \rightarrow \) 7 operations
- 4 rings \( \rightarrow \) 15 operations

**Cost:** \( 2^{N-1} = O(2^N) \)

- 64 rings \( \rightarrow \) \( 2^{64} \) operations
Towers of Hanoi

- \texttt{hanoi(from,to,other,number)}
  \begin{itemize}
  \item \texttt{-- move the top number disks}
  \item \texttt{-- from needle from to needle to}
  \end{itemize}

\textbf{if} number=1 \textbf{then}

\begin{itemize}
  \item move the top disk from needle from to needle to
\end{itemize}

\textbf{else}

\begin{itemize}
  \item hanoi(from,other,to, number-1)
  \item hanoi(from,to,other, 1)
  \item hanoi(other,to, from, number-1)
\end{itemize}

\textbf{end}

Some math that is good to know

- \( \log_b(xy) = \log_b x + \log_b y \)
- \( \log_b(x/y) = \log_b x - \log_b y \)
- \( \log_b x^a = a \log_b x \)
- \( \log_b a = \log_x a / \log_x b \)
- \( a^{b+c} = a^b a^c \)
- \( a^{bc} = (a^b)^c \)
- \( a^b / a^c = a^{(b-c)} \)
- \( b = a^{\log_a b} \)
- \( b^c = a^{c \log_a b} \)