Recursion Underpinnings

- Every instance of a function execution (call) creates an Activation Record, (frame) for the function.
  - Activation records hold required execution information for functions:
    - Return value for the function
    - Pointer to activation record of calling function
    - Return memory address, (calling instruction address)
    - Parameter storage
    - Local variable storage

Storage Corruption

- Infinite regression results in a collision between the “run-time” stack & heap termed a “run-time” stack overflow error.
- Illegal pointer dereferences (garbage, dangling-references)

Comparing Algorithms

Should we use Program 1 or Program 2? Is Program 1 “fast”? “Fast enough”?

The empirical approach

Implement each candidate → Run it → Time it
Running Time Implications

- Processor speed differences are too great to be used as a basis for impartial algorithm comparisons.
- Overall system load may cause inconsistent timing results, even if the same compiler and hardware are used.
- Hardware characteristics, such as the amount of physical memory and the speed of virtual memory, can dominate timing results.
- In any case, those factors are irrelevant to the complexity of the algorithm.

Analytical Approach

- Primitive operations
  - \( x = 4 \) assignment
  - \( ... x + 5 ... \) arithmetic
  - \( \text{if} (x < y) \) comparison
  - \( x[4] \) index an array
  - \( \text{"x} \) dereference
  - \( x.\text{foo( )} \) calling a method
- Others
  - \( \text{new/malloc} \) memory usage

Rules for Analysis

1. We assume an arbitrary time unit.
2. Running of each of the following type of statement takes time \( T(1) \):
   1. assignment statement
   2. I/O statement
   3. Boolean expression evaluation
   4. function return
   5. arithmetic operations
3. Running time of a selection statement (if, switch) is \( T(1) \) for the condition evaluation + the maximum of the running times for the individual clauses in the selection.

More Rules

4. Loop execution time is the time for the loop setup (initialization & setup) + the sum, over the number of times the loop is executed, of the body time + time for the loop check and update operations.
5. Running time of a function call is \( T(1) \) for function setup + the time required for the execution of the function body.
6. Running time of a sequence of statements is the largest time of any statement in the sequence.

Summation Formulae

- \( \sum_{k=1}^{N} Cf(k) = C \sum_{k=1}^{N} f(k) \) [S1: factor out constant]
- \( \sum_{k=1}^{N} C = NC \) [S3: sum of constant]
- \( \sum_{k=1}^{N} (f(k) \pm g(k)) = \sum_{k=1}^{N} f(k) \pm \sum_{k=1}^{N} g(k) \) [S2: separate summed terms]
- \( \sum_{k=1}^{N} k = \frac{N(N+1)(2N+1)}{6} \) [S4: sum of \( k \)]
- \( \sum_{k=1}^{N} k^2 = \frac{N(N+1)}{2} \) [S5: sum of \( k^2 \)]

How many foos?

```java
for (j = 1; j <= N; ++j) {
    foo();
}
```

\[ \sum_{j=1}^{N} 1 = N \]
How many foos?

for (j = 1; j <= N; ++j) {
    for (k = 1; k <= M; ++k) {
        foo( );
    }
}

\[ \sum_{j=1}^{N} \sum_{k=1}^{M} 1 = NM \]

How many foos?

for (j = 1; j <= N; ++j) {
    for (k = 1; k <= \( j \); ++k) {
        foo( );
    }
}

\[ \sum_{j=1}^{N} \sum_{k=1}^{j} 1 = \sum_{j=1}^{N} j = \frac{N(N+1)}{2} \]

How many foos?

void foo(int N) {
    if(N <= 2)
        return;
    foo(N / 2);
}

T(0) = T(1) = T(2) = 1
T(n) = 1 + T(n/2) if n > 2
= 2 + (1 + T(n/4))
= 3 + (1 + T(n/8))
= 4 + (1 + T(n/16))
...
\approx \log_2 n

How many foos?

for (j = 0; j < N; ++j) {
    for (k = 0; k < \( j \); ++k) {
        foo( );
    }
}

int N=M;
for (j = 0; j < N; ++j) {
    for (k = 0; k < M; ++k) {
        foo( );
    }
}

\[ N(N+1)/2 \quad \left\{ \begin{array}{l}
                         N^2 \\
                       N(N+1)/2
                       \end{array} \right. \]

Estimate: \( f(n) = 3n^2 + 5n + 100 \)

<table>
<thead>
<tr>
<th>n (input size)</th>
<th>time</th>
<th>( f(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>41</td>
</tr>
<tr>
<td>7</td>
<td>70</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>90</td>
<td>71</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>82</td>
</tr>
</tbody>
</table>

Function Estimation

If \( n \geq 10 \) then \( n^2 \geq 100 \)
If \( n \geq 5 \) then \( n^2 \geq 5n \)
Therefore, if \( n \geq 10 \) then:
\( f(n) = 3n^2 + 5n + 100 < 3n^2 + n^2 + n^2 = 5n^2 \)

So \( 5n^2 \) forms an “upper bound” on \( f(n) \) if \( n \) is 10 or larger (asymptotic bound). In other words, \( f(n) \) doesn't grow any faster than \( 5n^2 \) “in the long run”.

3
Big-Oh Defined

- To say $f(n)$ is $O(g(n))$ is to say that $f(n)$ is “less than or equal to” $Cg(n)$
- More formally, let $f$ and $g$ be functions from the set of integers (or the set of real numbers) to the set of real numbers. Then $f(x)$ is said to be $O(g(x))$, which is read as $f(x)$ is big-oh of $g(x)$, if and only if there are constants $C$ and $n_0$ such that $|f(x)| \leq C \cdot |g(x)|$ whenever $x > n_0$.
- Don’t be confused … “$f(n)$ is of Order $g(n)$”

The trick

$$N(N+1)/2 + N^2$$

$$an^k + bn^{k-1} + \ldots + yn + z$$

Some complexity classes …

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Big-Oh Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Linear</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Quadratic</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Cubic</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>Polynomial</td>
<td>$O(n^p)$</td>
</tr>
<tr>
<td>Exponential</td>
<td>$O(a^n)$</td>
</tr>
</tbody>
</table>

Does it matter?

Let $n = 1,000$, & 1 ms / operation.

<table>
<thead>
<tr>
<th>$n$</th>
<th>Max $n$ in one day (first day)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>1 second</td>
</tr>
<tr>
<td>$n \log_2 n$</td>
<td>10 seconds</td>
</tr>
<tr>
<td>$n^2$</td>
<td>17 minutes</td>
</tr>
<tr>
<td>$n^3$</td>
<td>12 days</td>
</tr>
<tr>
<td>$n^4$</td>
<td>32 years</td>
</tr>
<tr>
<td>$n^{10}$</td>
<td>$3.17 \times 10^{19}$ years</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$1.07 \times 10^{301}$ years</td>
</tr>
</tbody>
</table>

Practical Curves

Common Growth Curves

Practical Curves

Low-order Curves

Log n

log n

$n^n$

$n \log n$
**Example**

- **Assume:**
  - 1 day = 100,000 sec (10^5)
  - (actually 86,400)
  - Input size $n = 10^6$
  - A computer that executes 1,000,000 (10^6) Instructions/sec
  - C/C++ statement instructions

**Question?**

- Does the fact that hardware is always becoming faster hardware mean that algorithm complexity doesn’t really matter?
- Suppose we could obtain a machine that was capable of executing 10 times as many instructions per second (so roughly 10 times faster than the machine hypothesized on the previous slides).
- How long would the order $n^2$ algorithm take on this machine with an input size of $10^6$?

**Big-Oh Defined**

- To say $f(n)$ is $O(g(n))$ is to say that $f(n)$ is “less than or equal to” $Cg(n)$
  - Where $C$ is some constant
  - $f(n) = 3n^2 + n + 5$
  - $C=5$
  - $g(n)=n^2$
- Don’t be confused …
  - “$f(n)$ is of Order $g(n)$”

**Doing the Numbers**

- **Order: $n^2$**
  - # instructions: $(10^6)^2 = 10^{12}$
  - # seconds to run: $10^{12} / 10^8 = 10^4$
  - # days to run: $10^4 / 10^5 = 1$

- Impressed? You shouldn’t be. That’s still 1 (instead of 20) day versus 20 seconds if an algorithm of order $n \log_2 n$ were used.
- What about 100 times faster hardware? 2.4 hours.

**Comparison**

- **Order: $n^2$**
  - # instructions: $(10^6)^2 = 10^{12}$
  - # seconds to run: $10^{12} / 10^8 = 10^4$
  - # days to run: $10^4 / 10^5 = 1$

- What about $n^2 + c^2$? What about $n^2 + n$? What about $cn^2$?
Observations

- Within complexity classes the differences between algorithms due to constants of proportionality, (coefficients & lesser terms), are not significant enough to warrant reporting.
- Exception: certain (high usage) helper algorithms (e.g., sorting, searching)
  - Because they are used many times
  - Think about a trip to NOVA with cars that drive 60 and 70+ mph respectively, one trip vs. weekly trips.

Observations

- Even for moderately small input sizes, Order n^2 algorithms will require FAR more time than Order n log(n) algorithms.
- Large problems with Order > n log(n) cannot practically be executed
  - For n = 1000 (medium problems) n^2 algorithms can still be used.

General Rules

- A normal loop has big Oh, O(n)
- A doubly nested loop has big Oh, O(n^2)
- A triply nested loop has big Oh, O(n^3)
- You can get better times, e.g. O(log n)
  - Binary Search is O(log n)
  - Anytime anything is halved on each iteration, you usually get O(log n)
- Why isn't Merge Sort O(log n)?

The trick

\[
\frac{N(N + 1)}{2} + N^2 + an^k + bn^{k-1} + \ldots + yn + z
\]

Best Case Analysis

- Assumes the input, data etc. are arranged in the most advantageous order for the algorithm, i.e. causes the execution of the fewest number of instructions.
  - E.g., sorting - list is already sorted; searching - desired item is located at first accessed position.

Worst Case Analysis

- Assumes the input, data etc. are arranged in the most disadvantageous order for the algorithm, i.e. causes the execution of the largest number of statements.
  - E.g., sorting - list is in opposite order; searching - desired item is located at the last accessed position or is missing.
Average Case Analysis

- Determines the average of the running times over all possible permutations of the input data.
  - E.g., searching - desired item is located at every position, for each search), or is missing.

**Big-Omega**

- Definition: Let $f$ and $g$ be functions from the set of integers (or the set of real numbers) to the set of real numbers. Then $f(x)$ is said to be $\Omega(g(x))$, which is read as $f(x)$ is big-omega of $g(x)$, if there are constants $C$ and $n_0$ such that $|f(x)| \geq C|g(x)|$ whenever $x \geq n_0$.
- Finds order of “best case”

**Big-Theta**

- Definition: Let $f$ and $g$ be functions from the set of integers (or the set of real numbers) to the set of real numbers. Then $f(x)$ is said to be $\Theta(g(x))$, which is read as $f(x)$ is big-theta of $g(x)$, if $f(x)$ is $O(g(x))$, and $\Omega(g(x))$.
- In other words, if Big Oh = Big Omega

**Example Of Big Theta**

Consider the function $f(x) = 5x^3 + x^2 + 1/(1+x^2)$. Without going through the complete details on the proof, it’s apparent that $f$ is $O(x^3)$, since $f(x) \leq 7x^3$ for $x \geq 1$.
- $f$ is $\Omega(x^3)$, since $f(x) \geq 5x^3$ for $x \geq 1$.
- Hence $f$ is both $O(x^3)$ and $\Omega(x^3)$, and thereby $f$ is $\Theta(x^3)$ also.

**Big Oh?**

```cpp
//We know N>M
for (j = 1; j <= N; ++j) {
    //foo( );
    for (k = 1; k <= M; ++k) {
        foo( );
    }
}
```

$$\sum_{j=1}^{N} 1 = N$$  
$$\sum_{j=1}^{N} \sum_{k=1}^{M} 1 = O(NM) = O(N^2)$$
Big Oh?

for (j = 1; j <= N; ++j) {
    for (k = 1; k <= j; ++k) {
        foo( );
    }
}

\[
\sum_{j=1}^{N} \sum_{k=1}^{j} 1 = \sum_{j=1}^{N} j = \frac{N(N+1)}{2} = O(n^2)
\]

Big Oh?

\[
\sum_{j=1}^{N} \sum_{k=1}^{j} 1 = \sum_{j=1}^{N} j = \frac{N(N+1)}{2} = O(n^2)
\]

for (j = 0; j < N; ++j) {
    for (k = 0; k < j; ++k) {
        foo( );
    }
}

int foo(int N) {
    if(N <= 2) return 0;
    return foo(N / 2);
}

T(0) = T(1) = T(2) = 1
T(n) = 1 + T(n/2) if n > 2

≈ O(log₂ n)

Big Oh?

for (j = 0; j < N; ++j) {
    for (k = 0; k < M; ++k) {
        foo( );
    }
}

int N=M;
for (j = 0; j < N; ++j) {
    for (k = 0; k < j; ++k) {
        foo( );
    }
}

int foo(int N) {
    if(N <= 2) return 0;
    return foo(N / 2);
}

T(0) = T(1) = T(2) = 1
T(n) = 1 + T(n/2) if n > 2

≈ O(log₂ n)