Running Time Analysis of Program Code

Model for Computation

- To perform running time analysis on program code (C++ code, for e.g.), we need to adopt a model for computation
  - Tells how to deal with the running time requirements of certain basic (elementary, primitive) operations
- We'll use a very simple model for our (introductory) study:
  - Each basic operation on all basic types take 1 unit of time
    - Arithmetic (+, −, `*`, `/`, `%`, +=, etc.), comparison (==, <, >, !=, <=, etc.), assignment (=), pointer dereference(`*`), indexing into an array, calling a function, returning from a function, etc. each takes 1 unit of time when applied to `int`, `float`, `double`, `char` and other basic types
    - However, for e.g., addition of `vector<int>` (a non-basic type) doesn't take 1 unit of time but will depend on the size of the vector involved

Running Time Analysis of Program Code

Simplifying Rules ("formally")

- Some math properties of Big-O that help:
  - If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$ — *I.e.*, $O(O(g(n)))$ is $O(g(n))$
  - If $f(n)$ is $O(k \times g(n))$ for any constant $k>0$, then $f(n)$ is $O(g(n))$ — If $f(n)$ is $k \times g(n)$ then $f(n)$ is $O(g(n))$
  - If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(max(g_1(n), g_2(n)))$
  - If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n)\, f_2(n)$ is $O(g_1(n)\, g_2(n))$
  - The function $log_a n$ is $O(log_b n)$ for any positive number a and b not equal to 1

Running Time Analysis of Program Code
Simplifying Rules (effectively)

- Due to preceding math properties of Big-O:
  1) We *can ignore low-order terms* in an algorithm's growth rate function
     - E.g.: If an algorithm is $O(n^3 + 4n^2 + 3n)$, it is also $O(n^3)$
  2) We *can ignore a multiplicative constant* in an algorithm's growth rate function
     - E.g.: If an algorithm is $O(5n^3)$, it is also $O(n^3)$
  3) We *can combine* an algorithm's growth rate functions
     - E.g.: $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
       Thus, if an algorithm is $O(n^2) + O(n)$, it is also $O(n^2 + n)$, which we can more simply write as $O(n^2)$ by applying ***Rule 1***

Running Time Analysis of Program Code
Rule for Analyzing Consecutive Statements

- Time to execute consecutive statements is the sum of the times to execute the individual statements
- For e.g., consider following code segment:

```
S1; // takes T1
S2; // takes T2
S3; // takes T3
```

  - Here `S1`, `S2`, `S3` each could be any <u>*set*</u> of statements
  - Total time to execute the statements is `(T1+T2+T3)`

Running Time Analysis of Program Code

Rule for Analyzing Loops

- A loop involving *n* iterations can be viewed as *n* consecutive statements → *loop unfolding*
- For e.g., consider following code segment:

```
for (i = 1; i <= n; ++i)
    Si; // takes Ti
```

  - Here `Si` is any <u>*set*</u> of statements that will be executed during the *i*-th iteration of the loop
  - Total time to execute loop is Σ_i `(Ti)` plus the time it takes to initialize the counter and to increment it during execution

Running Time Analysis of Program Code

Rule for Analyzing Nested Loops

- Analyze the loops *from the innermost out*
- For e.g., consider following code segment:

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= n; ++j)
        Sij; // takes Tij
```

  - Here `Sij` is any <u>*set*</u> of statements that will be executed during the *j*-th iteration of the inner loop within the *i*-th iteration of the outer loop
  - Inner loop takes time `Ti` = Σ_j `(Tij)`
  - Total time to execute nested loop is Σ_i `(Ti)` = Σ_i `(`Σ_j `(Tij))` plus the time it takes to initialize the counters and to increment them during execution

Running Time Analysis of Program Code

Rule for Analyzing `if…else` *Type Statements*

- We will use the running time for the *worst case*
- For e.g., consider following code segment:

```
if (C1)
   S1; // takes T1
else if (C2)
   S2; // takes T2
...
else
   Sk; // takes Tk
```

  - Here `Ci` is any appropriate condition, and `Si` could be any *set* of statements
  - Total time to execute statement is $\mathtt{max}_i\mathtt{(Ti)} + \mathtt{k}$

Math Review: Summations

- **Some useful summations:**

$$\sum_{i=1}^{n} i = n(n+1)/2$$

$$\sum_{i=1}^{n} i^2 = (2n^3 + 3n^2 + n)/6$$

$$\sum_{i=1}^{\log n} n = n \log n$$

$$\sum_{i=0}^{\infty} a^i = 1/(1-a) \quad \mathit{for} \quad 0 < a < 1$$

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

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \quad \mathit{for} \quad a \neq 1$$

Math Review: Logarithms and Exponents

- **Definition:**

$$\log_b a = c \quad \text{if} \quad a = b^c$$

- **Some important rules:**

$$\log_b ac = \log_b a + \log_b c \qquad b^{\log_c a} = a^{\log_c b}$$

$$\log_b a/c = \log_b a - \log_b c \qquad (b^a)^c = b^{ac}$$

$$\log_b a^c = c \log_b a \qquad b^a b^c = b^{a+c}$$

$$\log_b a = (\log_c a)/\log_c b \qquad b^a/b^c = b^{a-c}$$

Running Time Analysis of Program Code

- *Example 1a:*

```
x = y;
```

- *Example 1b:*

```
void print_hello()
{
    cout << "Hello!\n";
}
```

Running Time Analysis of Program Code

- *Example 2a (summing loop):*

```
sum = 0;
while (i = 1; i <= n - 1; ++i)
   sum += n;
```


Running Time Analysis of Program Code

- *Example 2b (copying a C-string to another):*

```
void CstrCpy(char t[], const char s[])
{
   int i = 0;
   while ( s[i] != '\0' )
   {
      t[i] = s[i];
      ++i;
   }
   t[i] = '\0';
}
```

Running Time Analysis of Program Code

- *Example 3 (outputting a square matrix):*

```
void CoutSqMatrix(int mat[][n], int n)
{
   int row, col;
   for(row = 0; row < n; ++row)
   {
      for(col = 0; col < n; ++col)
         cout << mat[row][col]
   }
}
```


Running Time Analysis of Program Code

- *Example 4a:*

```
for (i = 0; i < n; ++i)
{
   sum = a[0];
   for (j = 1; j <= i; ++j)
      sum += a[j];
   cout << "Sum of subarray 0 through "
        << i << " is " << sum << endl;
}
```

Running Time Analysis of Program Code

- *Example 4b:*

```
for (j = 1; j <= n; ++j)
{
   k = 1;
   while ( k <= n)
   {
      k = 2 * k;
   }
}
```


Running Time Analysis of Program Code

- *Example 5a:*

```
sum1 = 0;
for (k = 1; k <= n; k *= 2)
   for (j = 1; j <= n; ++j)
      ++sum1;
```

Running Time Analysis of Program Code

- *Example 5b:*

```
sum2 = 0;
for (k = 1; k <= n; k *= 2)
   for (j = 1; j <= k; ++j)
      ++sum2;
```


Running Time Analysis of Program Code *(E.g. 5b Extra)*

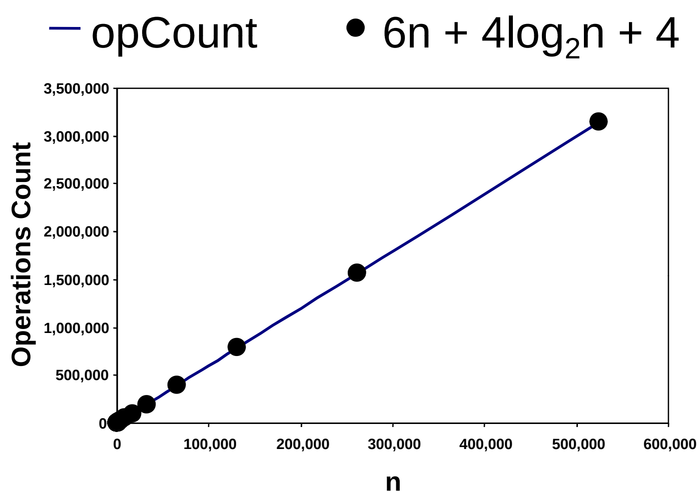
```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{   unsigned long int j, k, n = 1, opCount, sum2;

    while (n <= 1000000)
    {   opCount = 0;
        sum2 = 0;
        ++opCount;
        k = 1;
        ++opCount;
        for (; k <= n; k *= 2)
        {   opCount += 2;
            j = 1;
            ++opCount;
            for (; j <= k; ++j)
            {   opCount += 2;
                ++sum2;
                ++opCount;
            }
            ++opCount;
        }
        ++opCount;
        cout << n << '\t' << opCount << endl;
        n *= 2;
    }

    return 0;
}
```

Running Time Analysis of Program Code *(Example 5b Extra)*



Textbook Readings

- Pages 22-23
  - Time Analysis of C++ Functions
- Pages 117-118
  - The Bag Class – Analysis
- (There are also analyses of specific algorithms appearing at various places when the algorithms are introduced and discussed. Add these to the list as and when the algorithms are covered.)