Announcements

- Program due tomorrow
- Next HW is posted, due 1 week from tomorrow (make sure the code works!)
- Review session Feb. 28 (Monday)
- Midterm on Tuesday, March 1
How to measure speed of a program?

Counting primitive operations

Identify high-level primitive operations independent of language compilers OS, or computer

Operations:
- create variables
- variable assignment
- comparison
- addition
- multiplication...
- boolean operations

\[ x = x + 1 \]
Counting operations: (pseudocode)

Algorithm arrayMax(A, n):
    Input: An array A of n ≥ 1 numbers
    Output: The maximum element of A

1. currentMax ← A[0] ← 1
2. for i ← 1 to n−1 ← repeats n−1 times
3. if currentMax < A[i] then ← 1 comparison
4.     currentMax ← A[i] ← 1 assignment
5. return currentMax ← 1 return

best: 1 + (n-1)1 + 1 = n + 1
worst: 1 + (n-1)2 + 1 = 2n
Asymptotic Notation - Ch. 4

How important is exact number of computations?

In general, any primitive statement depends on a small number of low-level operations, independent of language or computer.

So we'll focus on big-picture, or how the running time grows in proportion to input size (usually $n$).
Formalize: Big-Oh notation

Let $f(n)$ and $g(n)$ be two functions from non-negative integers to reals.

We say $f(n)$ is $O(g(n))$ if there exists a constant $c$ and integer $N_0 > 0$ such that $f(n) < c \cdot g(n)$ for all $n \geq N_0$.

$f(n)$ is big-Oh of $g(n)$
Ex: \( f(n) \rightarrow g(n) \)  
\[ 4n - 2 \text{ is } O(n). \]

Why? Find \( c \) and \( n_0 \), s.t. \( \forall n > n_0 \), 
\[ 4n - 2 < c \cdot n \]
Let \( c = 5 \), \( n_0 = 0 \)
Ex: running time of arrayMax is \( O(n) \):

Algorithm arrayMax(A, n):
Input: An array A of \( n \geq 1 \) numbers
Output: The maximum element of A

\[
currentMax \leftarrow A[0]
\]
for \( i \leftarrow 1 \) to \( n-1 \):
if \( currentMax < A[i] \) then
\[
currentMax \leftarrow A[i]
\]
return \( currentMax \)

\[2n \text{ is } O(n)\]
Ex: \(26n^3 + 10n \log n + 5\)

is \(O(n^3)\)

\(20n^3 + 10n \log_2 n + 5 \leq 35n^3\)

so let \(c = 35, \ n_0 = 1\)
Any polynomial: \( f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \)

highest power is what we care about

so \( f(n) = \Omega(n^k) \)
Ex: $2^{100}$ is $O(1)$

Let $c = 2^{100} + 1$

Here: $O(1)$ - constant time

$O(n)$ - linear time (for loops)

$O(n^2)$ - quadratic time
Ch 4 in book: Rules & Examples

For any data structure, we’ll provide big-O bounds on our functions.

This is one way to compare which data structure will work best.
Useful things to remember:

\[ \sum_{i=a}^{b} f(i) = f(a) + f(a+1) + \ldots + f(b) \]

(Loops often produce these!)

Example:

\[ \text{for } (\text{int } i=0; i<n; i++) \]

\[ x = x+1 \]

or

\[ \text{for } (\text{int } i=0; i<n; i++) \]

\[ A[i][j] = i \cdot j \]

\[ \sum_{i=0}^{n} \frac{n}{i} = n \]

\[ \sum_{i=0}^{n-1} 2 = 2 + 2 + \ldots + 2 = 2n \]

\[ \text{n times} \]
Another:

\[ \sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \frac{n \cdot (n+1)}{2} = \Theta(n^2) \]

When might this come in handy?

for \( i = 1 \) to \( n \)

for \( j = 1 \) to \( i \)

\( x = x + j \)
Stacks and queues (array-based)

**Stack**
- `push(e)` : $O(1)$
- `pop` : $O(1)$
- `size` : $O(1)$
- `empty` : $O(1)$
- `destructor` : $O(1)$

**Queue**
- `push(e)` : $O(1)$
- `pop()` : $O(1)$
- `size` : $O(1)$
- `empty` : $O(1)$

J data structure (in terms of speed)
**Singly Linked List:**

- **insert Front (e):** $O(1)$
- **remove Front (·):** $O(1)$
- **empty:** $O(1)$
- **destructor:** $O(n)$

```c
while (!empty())
    loop repeats n times
    remove Front() \(\rightarrow O(1)\)
```
Linked queues

head

STL → MSP → ATL

insert here

delete here

not going to use SLinkedList (or DLinkedList)
template <typename Object>
class LinkedQueue {

    private:

        class QNode {
            private:
                Object *data;
                QNode *next;
        }

    QNode *head;
    QNode *tail;
    int size;

    more on Monday!