505 22240 / ESOE 2012 Data Structures: Lecture 6
Asymptotic Analysis, Stacks, and Queues

§ Asymptotic Analysis (bounds on running time or memory)

・Knowing the complexity of algorithms allows you to answer questions such as

— How long will a program run on an input? → Time Complexity

— How much space (storage) will it take? → Space Complexity

— Is the problem solvable?

・Asymptotic analysis is based on the idea that as the problem size grows, the complexity can be described as a simple proportionality to some known function.

・Suppose an algorithm for processing inventory management of a retail store:

— 10,000 ms to read inventory from disk.

— 10 ms to process each transaction.

➔ n transactions takes (10,000 + 10n) ms.

・<u>10n</u>: more important if n is large even though 10,000 >> 10.

・We want a way to express the speed of an algorithm independently of a specific implementation on a specific machine.

◎Big-Oh Notation (upper bounds on a function's growth)

・We use Big-Oh notation to say how slowly code might run as its input grows.

・Let n be the size of a program's <u>input</u> (e.g., bits, data words, …).

・Let T(n) be function,. e.g., running time.

・Let f(n) be another function → preferably simple. (e.g., f(n) = n).

・T(n) ∈ O(f(n)) IF&ONLY IF T(n) ≤ cf(n) whenever n is **BIG** for some **LARGE** CONSTANT c.

※How big is BIG?

➔ Big enough to make T(n) fit under cf(n).

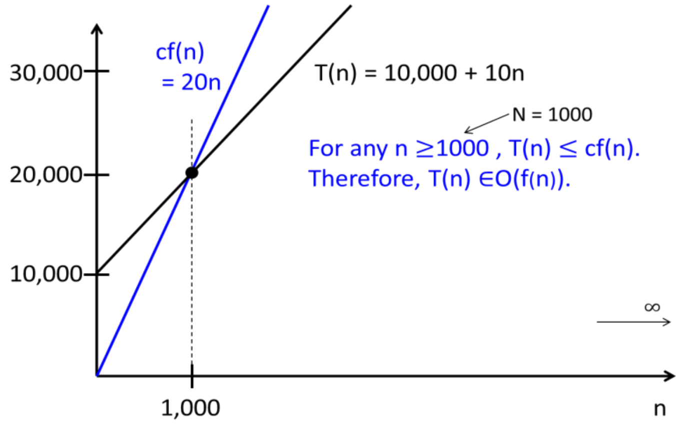
※How large is LARGE?

➔ Large enough to make T(n) fit under cf(n).

・e.g. <u>Inventory management</u>

T(n) = 10,000 + 10n

Let's try f(n) = n. Pick c = 20.



◎Formally description: Big-Oh

O(f(n)) is the SET of ALL functions T(n) that satisfy: There exist positive constants c and N such that, for all n ≥ N, T(n) ≤ cf(n).

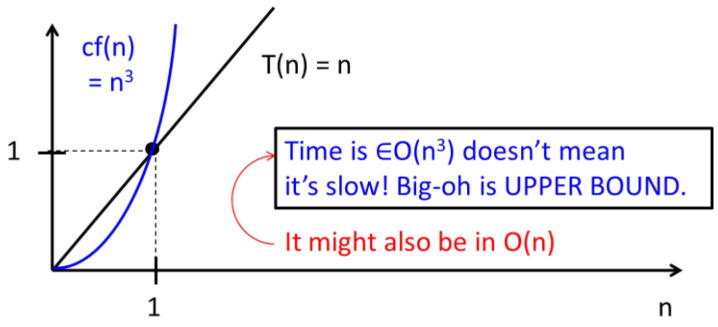
◎Examples:

① 1,000,000n ∈ O(n) Proof: choose c = 1,000,000, N = 0.

➔ <u>Big-Oh</u> notation doesn't care about (most) constant factors.

➔ Generally leave constants out. Unnecessary to write O(2n)

② n ∈ $O(n^3)$ Proof: set c = 1, N = 1.



③ $n^3+n^2+n \in O(n^3)$ Proof: set c = 3, N = 1.

➔ Big-oh notation is usually used to indicate the dominating (fastest-growing) (largest) term.

◎Table of important Big-Oh sets

・Smallest to largest:

Function	Common name
$O(1)$	constant
$\subset O(\log n)$	logarithmic
$\subset O(\log^2 n)$	log-squared
$\subset O(\sqrt{n})$	root-n
$\subset O(n)$	linear
$\subset O(n \log n)$	n log n
$\subset O(n^2)$	quadratic
$\subset O(n^3)$	cubic
$\subset O(n^4)$	quartic
$\subset O(2^n)$	exponential
$\subset O(e^n)$	exponential (but more so)
$\subset O(n!) \subset O(n^n)$	

・O(n log n) or faster time: considered "efficient".

・n^7 or slower time: considered useless.

◎Warnings

① Fallacious proof:

$n^2 \in O(n)$, Proof: choose c = n, Then $n^2 \leq n^2$.

➔ Wrong! c must be a constant. (cannot depend on n)

② Big-Oh notation DOES NOT SAY What the functions are. (it expresses a relationship between functions)

・e.g. Binary search on an array:

➔ Worst-case running time $\in$ O(log n)

➔ Best-case running time $\in$ O(1)

➔ Memory use $\in$ O(n)

➔ 47 + 18log n – 3 / n $\in$ O(the worst-case running time)

③ $e^{3n} \in O(e^n)$ because constant factors don't matter.
$10^n \in O(2^n)$ because constant factors don't matter.

⇨Wrong!

✓ e^{3n} is bigger by a factor of e^{2n}.

✓ 10^n is bigger by a factor of 5^n.

④ Big-Oh notation doesn't always tell whole story.

・e.g. $T(n) = n \log_2 n$

$U(n) = 100n$

➔ T(n) dominates U(n) asymptotically. However, $\log_2 n < 50$ in practice.

➔ U(n) is only faster than T(n) if the input size is infinitely large.

◎Big-Omega

> Ω(f(n)) is the set of all functions that satisfy: There exist positive constants d & N such that, for all n $\geq$ N, T(n) $\geq$ df(n).

➔ Compare with Big-Oh: T(n) $\leq$ cf(n)

・ Ω is the reverse of Big-Oh:

If T(n) $\in$ O(f(n)), f(n) $\in$ Ω(T(n)) and vice versa.

・e.g.

$2n \in \Omega(n)$ BECAUSE $n \in O(2n)$

$n^2 \in \Omega(n)$ BECAUSE $n \in O(n^2)$

$n^2 \in \Omega(3n^2+n\log n)$ BECAUSE $3n^2+n\log n \in O(n^2)$

・Big-Omega gives a LOWER BOUND on a function.

・Big-Oh says "Your algorithm is at least this good."

・Big-Omega says "Your algorithm is at least this bad."

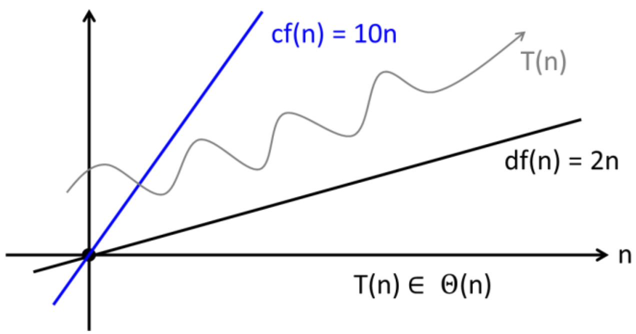
❋If T(n) $\in$ O(f(n)) and $\in$ Ω(g(n)), T(n) is sandwiched between cf(n) & dg(n).

➔ If f(n) = g(n), we say that T(n) $\in$ Θ(f(n))

◎Big-Theta

> Θ(f(n)) is the set of all function T(n) that are in both O(f(n)) and Ω(f(n)).

・e.g.



T(n) $\in$ Θ(n)

・Big-Theta is symmetric:

If f(n) $\in$ Θ(g(n)), then g(n) $\in$ Θ(f(n)).

・e.g.

$n^3 \in \Theta(3n^3-n^2)$, $3n^3-n^2 \in \Theta(n^3)$

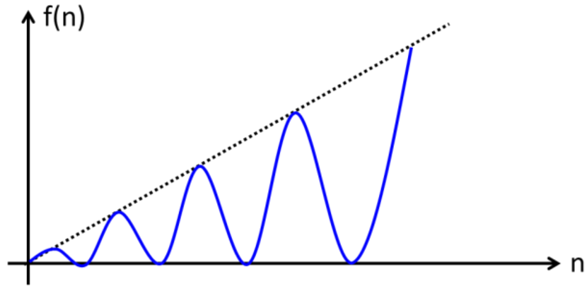
$n^3 \notin \Theta(n)$, $n \notin \Theta(n^3)$

$n \in O(n^8)$, n is NOT in $\Theta(n^8)$

・e.g. f(n) = n(1 + sin n) $\in$ O(n)

$\in$ Ω(0)

➔ But NOT in Θ(n) or Θ(0) (Not in "Theta" of anything simple)



・f(n) $\in$ Θ(g(n))

➔ f(n) $\in$ O(g(n))

➔ f(n) $\in$ Ω(g(n))

§ **Algorithm Analysis**

<u>Problem #1</u>: Give a set of p points, find pair closest to each other.

Algorithm #1: Calculate distance between every pair; return minimum.

➔ There are $\frac{p(p-1)}{2}$, each pair takes constant time to examine.

➔ Time $\in$ $\Theta(p^2)$, (worst- and best- case running times).

• code:

```
double minDistance = point[0].distance(point[1]);
/* Visit a pair (i, j) of points.*/
for (int i = 0; i < numPoints; i++) {
/* We require j > i so each pair is visited only once.*/
     for (int j = i+1; j < numPoints; j++) {
          double thisDistance = point[i].distance(point[j]);
          if (thisDistance < minDistance) {
               minDistance = thisDistance;
          }
     }
}
```

⇨ multiply → $\Theta(p^2)$ time

◎Functions of Several Variables

Problem #2: Matchmaking program for w women & m men.

Algorithm #2: Compare each man / each woman. Decide if they are compatible.

➔ Each comparison takes constant time.

➔ Running time $T(w, m) \in \Theta(wm)$.

⇨ There exist constants c, d, W & M such that $dwm \leq T(w, m) \leq cwm$ for every $w \geq W$, $m \geq M$.

T is NOT in $O(w^2)$, nor in $O(m^2)$, nor in $\Omega(w^2)$, nor in $\Omega(m^2)$.

➔ Every one of these possibilities is eliminated either by w >> m or m >> w.

You cannot asymptotically compare the functions wm, w^2, and m^2.

Problem #3: An array contains n music albums sorted by title. You request a list of albums starting with "The Best of …". Suppose there are k such albums.

Algorithm #3: Search for a match with binary search. Walk (in both directions) to find other matching albums.

★Worst-case time $\in \Theta(\log n + k)$.

• Binary search takes at most log n steps.

• Complete list of k matching is found, each in constant time.

• k can be as large as n, not dominated by log n.

• k can be as small as zero.

➔ No simpler expression.

• Algorithms like this are called "output-sensitive".

• Output-sensitive: Performance depends partly on size k of the output.

★Best-case time $\in \Theta(1 + k) = \Theta(k)$

• Binary search finds a match right away.

Problem #4: Find the k-th item in an n-node doubly-linked list

Algorithm #4: If k < 1 or k > n, report an error and return.

Otherwise, compare k with n - k.

If $k \leq n - k$, start at the beginning of the list and walk forward k - 1 nodes.

Otherwise, start at the end of the list and walk backward n - k nodes.

• If 1 <= k <= n, this algorithm takes $\Theta(\min\{k, n - k\})$ time (in all cases).

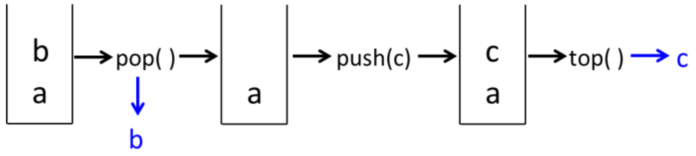
• This expression cannot be simplified: without knowing k and n, we cannot say that k dominates n - k or that n - k dominates k.

§ Stacks

- A stack is a crippled list.
- Only the item at the top be manipulated.
- A stack can grow arbitrarily large.
- A stack is a collection of objects that are inserted and removed according to the last-in first-out (LIFO) principle.

◎ Operations:

- ”push” new item onto top of stack.
- ”pop” top item off stack. → return and remove the top item.
- examine “top” item of stack. → return the top item, without removing it.
- e.g.



◎ Easily implemented as singly-linked list. All operations take O(1) time.

```
template <typename T>
class Stack {
public:
     int size( ) const;
     bool isEmpty( ) const;
     void push(const T& t); → insertFront( )
     T& pop( ); → removeFront( )     ┐
                                     ├ throw(EmptyStackException)
     const T& top( ); → front( )     ┘
};
```

◎ Sample application: Verifying matched parentheses in a string.

{ [( ) { [ ] }] ( ) }

⇨ Scan through the string, character by character.

- lefty — {, [, (: Push onto stack.
- righty : Pop its counterpart off stack; check that they match.

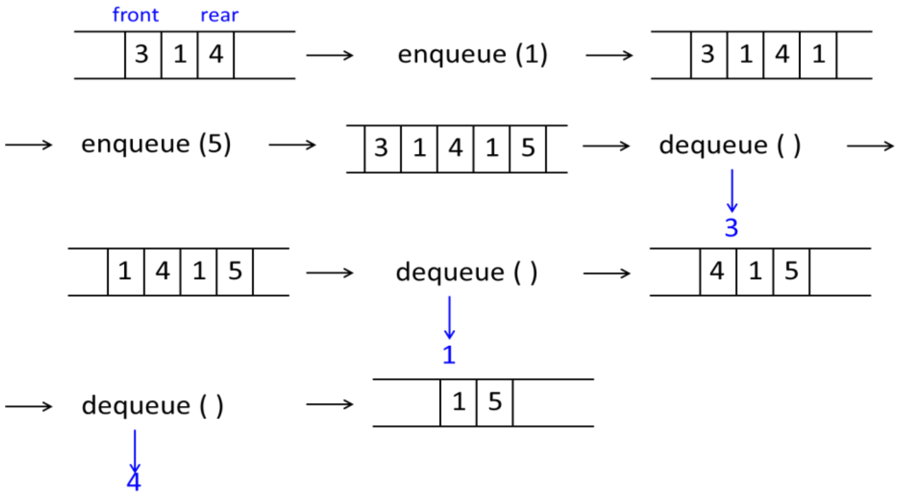
If <u>mismatch</u> or <u>try to pop empty stack</u> or <u>stack not empty at end of string</u>: parentheses not properly matched.

§ Queues

- A queue is also a crippled list.
- A queue is a collection of objects that are inserted and removed according to the first-in first-out (FIFO) principle.
- Items enter a queue at the <u>rear</u> (back) and are removed from the <u>front</u>.
- A queue can grow arbitrarily long.

◎ Operations

- ”enqueue”: insert item at rear of queue.
- ”dequeue”: remove and return the front item.
- ”front”: return the front item, without removing it.
- e.g.



・<u>Applications</u>: stores, theaters, reservation centers according to the FIFO principle.

§ Deques

・A deque (pronounced "deck") is a double-ended queue.

・Insert & remove items at both ends.

・Easily implemented as <u>doubly-linked</u> list.