Monday:
2^{nd} Midterm Exam

Algorithm analysis

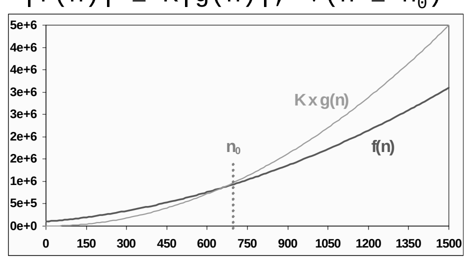
- Need a way to measure efficiency
  - Regardless of processor speed or compiler implementation
    - Both of which can *greatly* affect processing time
  - And independent of the programming language used
- Really just need a way to *compare* algorithms
  - i.e., holding constant things that don't matter
  - Question becomes – which algorithm is more efficient on *any computer in any language*?
- Solution – 'O' notation
  - Simplest type is worst case analysis – called Big-Oh
    - Little-oh, Big Ω (omega), and Big Θ (theta) – not in CS 12

Big-Oh notation

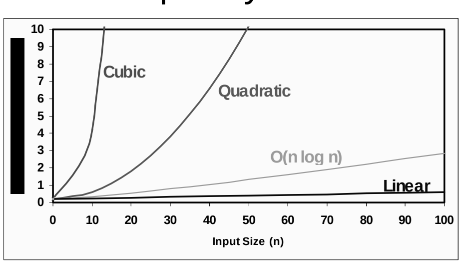
- Strips problem of inconsequential details
  - All but the "dominant" term are ignored
    - e.g., say algorithm takes `3n² + 15n + 100` steps, for a problem of size `n`
    - Note: as `n` gets large, first term (`3n²`) dominates, so okay to ignore the other terms
  - Constants associated with processor speed and language features are ignored too
    - In above example, ignore the `3`
- So this example algorithm is `O(n²)`
  - Pronounced "Oh of n-squared"
    - Belongs to the "quadratic complexity" class of algorithms

Formally, `f(n)` is `O(g(n))` if

$\exists$ two positive constants $(K,\ n_0)$, such that
$|f(n)| \leq K|g(n)|,\ \forall(n \geq n_0)$



Some complexity classes



- Linear - `O(n)`; Quadratic - `O(n²)`; Cubic - `O(n³)`
  - Also slower than cubic – e.g., Exponential - `O(2ⁿ)`
  - And faster than linear – `O(log n)`, and Constant - `O(1)`

Applies to large problems only

- Big-Oh measures asymptotic complexity
  - Mostly irrelevant for small problems
  - But some algorithms become impractical as n grows
- Say linear time is 256 microseconds (μsecs):
  - $O(\log_2 n)$ time is 8 μsecs
  - $O(n \log_2 n)$ time is 2.05 milliseconds (ms)
  - Quadratic time is 65.5 ms
  - Cubic time is 16.8 seconds
  - Exponential time (base 2) is 3.7×10^{63} years!!!

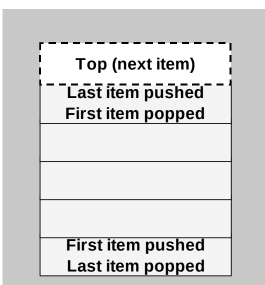
Efficiency of list functions

- If singly-linked list (like assignment 2):
  - Insert/delete first – O(1)
  - Insert/delete last/random – O(n)
    - If pointer to last item – insert last is O(1)
  - Find value – O(n)
  - Retrieve/set i^{th} item – O(n)
- Compare to array:
  - Insert/delete first/random, and find value – O(n)
  - Insert/delete last – O(1) – unless resize, then O(n)
  - Retrieve/set i^{th} item – O(1) – the array's strong point

What Big-Oh doesn't cover

- Small problems
  - Often dominated by lesser terms or constants
- What to count?
  - Comparisons? Assignments? Reads? Writes?
  - Some operations take longer than others
    - Depends in part on the system, compiler, and so on
- Notice the definition is not restrictive
  - e.g., an algorithm that is O(n) is also $O(n^2)$, etc.
  - So *agree* to express bound as tightly as possible, and to not include lesser terms in g(n)

Stacks



- LIFO data structure
  - Last In, First Out
- All items except last item pushed are inaccessible
- So has very few possible operations:
  - `push`, `pop`, `peek`, `empty`, `full`, `size`, `clear`
- Lots of applications

Applying stacks

- Can be used to eliminate recursion
  - Iteration and stacks instead of recursive calls
    - For each "recursive" step
      - Push `struct` full of critical data values
    - While stack is not empty
      - Pop `struct` – like "return" from recursive call
  - It's how the compiler does it
    - Pushes "activation record" (a.k.a., "stack frame") for every function call, not just recursive ones (see text section 7.7)
- In fact, idea applies to *any nested structure*
  - Recursion is just a nesting of function calls
  - What about nested parentheses in expressions?

Checking balanced ( ), [ ], { }

- Okay to nest, like `{x/[y*(a+b)]}`
- Not okay to mismatch (or nest improperly)
  - `(a/(x + y)` is missing a right parenthesis
  - `( x + [y-2)]` is mismatched at `[ )`
- Parentheses fully match if the following works:

```
for (each character in the expression) {
    if a left parenthesis - push it on the stack;
    if a right parenthesis
            pop matching left parenthesis from stack
} stack is empty at the end
```

- See program 7.5 in text

Implementing stacks

- Easy with a list (too easy for programming project):
  - Say `ListPointer list = createList();`
  - Then to push: `insertFirst(item, list);`
  - To pop: `return deleteFirst(list);`
  - To peek: `return firstInfo(list);`
  - To clear: `clearList(list);`
  - emptyStack: `return emptyList(list);`
  - fullStack: `return 0;` /* does not fill up */
- Easy with an array too
  - And it's more efficient – less function overhead