Week 3: Algorithm Efficiency

Performance measurement
Effect of problem size
Estimating worst and average cases
Big-O notation

Exact speed specification
- In the early days of large-scale computing (1956-1965) the documentation for using a library subroutine would often state exactly how long it would execute:

```
250 microseconds + N * 30 microseconds,
```

where $N$ is the size of the array

- Why was it practical to do that then?
- Why can't we do it today?

Efficiency as a function of problem size
- Many algorithms depend upon some factor, such as an array dimension or a list size.
  - We're interested less in the absolute speed than in how the running time increases as that factor increases.
  - It may turn out that an algorithm is intractible or impractical to run with more than a small set of data. Estimated run times in years or even centuries are not uncommon.
- Those algorithms are closely related to the data structures this course emphasizes.

The Big-O
- We say that a function $f(n)$ "has running time on the order of $g(n)$" or just
  $$f(n) \text{ is } O(g(n))$$
- For example a typical naive sort function will be $O(n^2)$ while a more sophisticated one is $O(n \log(n))$. 
Big-O arithmetic

- Is easy to figure.
- We don’t care about constant factors:
  - \(O(g(n) + C) = O(g(n))\)
  - \(O(Cg(n)) = O(g(n))\)
- Higher powers dominate (for large \(n\))
  - \(O(C_1n^3 + C_2n^2 + C_3n + C_4) = O(n^3)\).

Example

- aboveMeanCount (p. 106) is misnamed and misdocumented. Let’s clean it up:

  ```java
  // Count array elements exceeding a given value
  public static int countGreater(double[] array, double value) {
    int result = 0;
    for (int i=0, i < array.length, ++i)
      if (array[i] > value) ++result;
    return result; // No. of array elements > value
  }
  ```

- This is clearly \(O(a.length)\) or "linear in \(n\)."

A Java limitation?

- The previous example ought to be useful for arrays of other types:

  ```java
  // Count array elements exceeding a given value
  public static int countGreater(double[] array, double value) {
    int result = 0;
    for (int i=0, i < array.length, ++i)
      if (array[i] > value) ++result;
    return result; // No. of array elements > value
  }
  ```

- In C++ we’d make it a function template.
- In C we’d use a typedef or a macro.

A more complicated example

- Collins, p. 128-3.3 & p. 462

  ```java
  // Sort array of integers
  public static void sortint(int[] array) {
    for (int i = 0; i < array.length-1, ++i)
      for (int k = i+1, k < array.length, ++k)
        if (array[i] > array[k]) swap(array, i, k);
    return;
  }
  ```

- The performance analysis at the bottom of page 462 is valid.
- Why should we not call it selectionSort, as the textbook does?.
### Some common terminology for efficiency

- **O(1)**: Constant time
- **O(n)**: Linear in n
- **O(log n)**: Logarithmic in n
- **O(n log n)**: Linear-logarithmic in n
- **O(n^2)**: Quadratic in n
- **O(n^3)**: Cubic in n
- **O(n^k)**: Polynomial of degree k in n
- **O(2^n)**: Exponential in n

### One more example (cf. p. 112)

```java
// Binary search sorted table of long integers
// Returns index of tbl element that matches arg.
// If none, returns -k-1, where arg would lie
// between tbl[k] and tbl[k+1]

public static int btslong(long[] tbl, long arg) {
    int lb = 0, // lower bound
        hb = tbl.length-1; // upper bound
    while (lb <= hb) {
        int mid = (lb+hb) / 2; // Mid point
        if (arg == tbl[mid]) {
            return mid; // Found
        }
        if (arg < tbl[mid]) {
            lb = mid + 1; // Ignore lower half
        } else {
            hb = mid - 1; // Ignore upper half
        }
    }
    return -lb - 1; // Not found
}
```

### Questions about the binary search function

- Why was it acceptable to name the function `bts`, when we didn't approve of naming that sort function `selectionSort`?
- How many times will the loop execute
  - If a match is found?
  - If a match is not found?
- So what is big O for `btslong`?