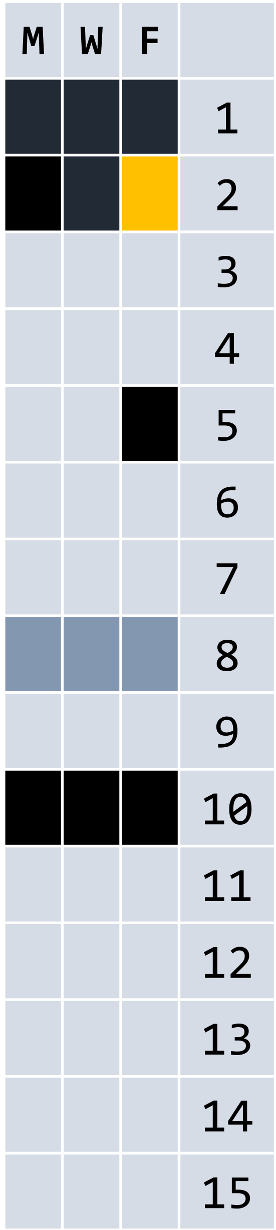
CMOR 421/521:

Complexity and Efficiency



Topics

What is performance?

- Complexity: Big O, Ω, Θ
  - Up-front constants
  - Lower order terms
  - Combinations of algorithms
  - Interdependence of time and memory complexity
- Implementation details
  - (joined by caching and stride from earlier)
  - Function overhead, inline functions, macros, and recursion

Algorithm Complexity

You may have heard people say:

- This algorithm is O(n*log(n))
- This program runs in polynomial time
- The problem is NP-hard/NP-complete
- This problem scales linearly
- This problem scales exponentially
- etc

Why Care about Complexity?

Complexity tells us how the cost of a program scales ***asymptotically*** *(i.e. if you go far enough out)*

- t = $: Computing time on a computer is often bought
- t = t: Solutions be needed by a deadline
- m = $: Space/memory on a computer is often bought
- m = m: You can run out of memory

Knowing the complexity of a program can help us estimate whether a run can complete a given dataset in a given amount of time and memory

Big O Mathematically

A given function f(X) is said to be O(g(X)) if there exists two constants k and C such that:

$$f(X) \leq C*g(X) \text{ for all } X > k$$

- Big O states an upper bound on a function
- Notice, by this definition n is O(n^2), O(n^3), etc
- We also require the upper bound to be ***asymptotically tight,*** i.e., as close as possible

Big O Practically

Example:

f(n) = 2*n^2 + 2*log(n) + n + 8

f(n) = ~~2*~~n^2 ~~+ 2*log(n) + n + 8~~

f is O(n^2)

- Ignore lower order terms
- Ignore coefficients

Big O Cheat Sheet

- Constant: .... O(1)
- Logarithmic: .... O(log(n))
- Log-linear/linearithmic: .... O(n*log(n))
- Polynomial: .... O(n^c)
- **Exponential:** .... O(a^n)
- **Factorial:** .... O(n!)
- **Tetration/combinatorial:** .... O(n^n)

Big O Practically: Example

Program:

For a square (n x n) 2D array:

1. Normalize the array by it’s maximum element
   1. Find the max
   2. Divide every element by the max
2. Search for an element in a given row, print the result
3. Print the max element and the normalized array

Big O Practically: Example

Example code: 1) Normalizing the array: 2*n^2

```
// Assume we're given: double A[n][n] = {…};

// Find the maximum element
double max = A[0][0];
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        if( A[i][j] > max)
            max = A[i][j];

// Normalize
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        A[i][j] /= max;
```

Big O Practically: Example

Example code: 2) Search for elements: n + 6

```
int i_search, j_query;

printf(“Enter the number to search for: ”);
scanf(“%lf”, query) )
printf(“Enter the row to be searched: ”);
scanf(“%d”, i_search);

j_query = -1;
for(j = 0; j < n && j_query < 0; j++)
    if A[i_search][j] == query
        j_query = j;

printf(“Query returned column %d\n”);
```

Big O Practically: Example

Example code: 3) Print results: n^2 + n + 1

```
// Print the max
printf("Max: %lf\n", max);

// Print the normalized array
for(i=0; i < n; i++) {
    for(j=0; j < n; j++)
        printf(" %.4lf", A[i][j]);
    printf("\n");
}
```

Big O Practically: Example

Program:

For a square (n x n) 2D array:

1. Normalizing: $2*n^2$
2. Search: $n + 6$
3. Print results: $n^2 + n + 1$

$$\Rightarrow f(n) = 3*n^2 + 2*n + 7 \Rightarrow O(n^2)$$

Note: Because we drop lower-order terms and coefficients, we typically don't count single actions (e.g. printing)

Big O Practically

Programs can rely on several variables/parameters

- The final Big O reported needs to include this
  - Previous example: What if we wanted to do multiple (q) searches?

    n-> q*n, => f would be O(n^2 + q*n), because q is unknown

Programs have both time and memory costs

- We can describe both using Big O notation
  - Previous example: Memory was O(n^2)
  - The memory costs do not (necessarily) go up if we do multiple searches

Big Ω: Big O's Opposite

A given function f(X) is said to be Ω(g(X)) if there exists two constants k and C such that:

$$f(X) \geq C*g(X) \text{ for all } X > k$$

- Big Ω states an **lower bound** on a function
- Notice, by this definition n^3 is Ω(n^2), Ω(n), …
- Big Ω must also be asymptotically tight
- For some functions, Ω is different than O

`Complexity != Efficiency`

Big O Isn’t Everything

Big O is a tool that helps us understand how a program will **scale**, but it neglects many things

- Algorithms behavior may vary on different datasets
  - Big O vs Big Ω
- What if two algorithms have the same complexity? Are they truly equal?
  - Their coefficients and lower order terms may be different
- Big O doesn’t know about architecture
  - Caching, stride, and fastest dimension/axis

Up-Front Constants

“Up-front constants” refers to the “C” in the mathematical statement: $f(X) \leq C*g(X)$ for $X > k$

- **Example:** suppose we want max, min, and average of an array
  - Can be calculate in one pass through O(n), or 3 separate passes O(3*n)
- Large up-front constants can make an algorithm with better complexity perform more poorly than an algorithm with more costly complexity for *smaller* datasets
- Remember Big O is asymptotic; ***eventually*** the better complexity algorithm will beat the other
  - What if you’re never encounter the cross-over point?
  - => **You choose the algorithm with higher (worse) complexity**

Combinations of Algorithms

- Sometimes a given task can be made more efficient by doing another task first
  - **Example:** Sorting before searching
  - Searching on unsorted data: O(n)
  - Searching on sorted data: O(log(n)), Ω(1)
  - Sorting, comparison based: O(n*log(n))
- **Previous example:** multiple (linear) queries was O(q*n)
  - If we sort the data first: O(q*log(n) + n*log(n))
  - Is that worth it? O(n*log(n)) is worse than O(n)…
  - Is q > n? If so, q*n > n^2 > n*log(n) => O(q*log(n))
  - For large number of queries (q > n), sorting will improve the runtime

Time and Memory Interdependency

- Big O can be used to describe an algorithm’s time complexity (how does runtime scale) and its memory complexity (how do memory needs scale)
- For some applications, we can trade time and memory complexity:
  - Faster algorithm, but more memory
  - Slower algorithm, but less memory
- **We’d like both complexities to be small, but often one is more constraining**
  - Examples: sparse matrices (prioritize memory), dynamic programming (prioritize time)

Big O and Architecture

Big O doesn’t know about architecture

- It also doesn’t know about the compiler and stack
- Recall: the amount of stack memory is “small”
  - But again, because the OS controls it, it doesn’t need to be big to be robust
- Every time a function is called, memory gets allocated on the stack
  - That process takes time that has nothing to do with runtime
  - **Function calls come with overhead costs**

Function Overhead

- **On the one hand:** We love functions
  - Functions make code more readable, debuggable, maintainable, reusable
- **On the other:** Functions come with overhead due to stack allocation and initialization
  - This is felt with recursive functions and in parallel
- **Readability/debuggability/maintainability, speed, and memory usage are all performance metrics**
  - "High Performance" is relative to the resources you have
- There are two methods for improving function overhead when it is an issue

Improving Function Overhead

- **Method 1: Macros**

```
#define cartesian2flat(i,j,n2) = i*n2 + j
```

- **Method 2: Inline functions**

```
inline int cartesian2flat(int I, int j, int n2) {…}
```

- Both behave like functions but macros can have type issues as they are precompiler commands
- These are used for functions that are: 1) very simple, and 2) called very often
- **Example:** indexing a mD array allocated as a 1D array

Next Up:
Memory and architecture

Recursive Functions

Recursive Functions: A function that calls itself

Example: $a_n = 2*a_{n-1}$, $a_0 = 1 \Rightarrow a_n = 2^n$

```
// This is a horrible excuse for recursion
void sequence(int n) {
    // Check if we're at the base case
    if( n == 0 )
        return 1;
    return(2*sequence(n-1));
}
// Better way for this example: iteratively
int result = 1;
for(i = 0; i < n; i++)
    result *= 2;
```

Recursion and Overhead

- Recursive functions are the typical culprits for stack overflow from a single program because they can be called many, many times (infinitely many if your base case is bad)
- This means they also incur greater overhead costs
- **Mathematical recursion does not imply programmatic recursion**
- Many mathematically recursive things can be programmed iteratively (i.e. with loops)
- **Prefer iterative implementations to recursive ones**
  - Sometimes recursion is the elegant solution! Use it then!

Performance

Program performance is dependent on several things

Performance depends on the quality of:

- Your resources => **Computing resources**
- Your solution to the problem => **Algorithm complexity**
- Your utilization of your resources => **Implementation details**

HPC != Supercomputer * bad solution * bad implementation

HP**C is about making the most of your computing resources, not having more resources**

Performance Checklist

- **Algorithm complexity:** How do you choose/design an algorithm?
  - **Priority:** What constraints do you have on time and memory? Is one more important/tightly constrained than the other?
  - **Parameters:** Are there competing parameters (q vs n in the prev. example) that need to be balanced?
  - **Domain:** Are your datasets big enough to justify the up-front constants of your choice?
- **Implementation details:** How did you implement the algorithm?
  - **Access patterns:** Is/can your algorithm/code friendly to caching?
  - **Function calls:** Do you minimize excessive function overhead?
  - **Efficiency:** Can you consolidate memory and traversals?
  - **Code quality:** Is your code as friendly as is needed?

Next Up:
Non-Communicating Parallelism

- **This is the end of the Pre-Parallelism section of the course**
  - We now have tools for/exposure to writing C++ code and evaluating a program's performance
  - **HW1 will be posted tonight**
- **Next week:** Non-communicating Parallelism **(OpenMP)**