CS 261-020
Data Structures

Lecture 5
Complexity Analysis
1/30/24, Tuesday
Odds and Ends

• Assignment 1 past due
  • 70% if submit by tonight

• Recitation 4 posted

• Assignment 2 will be posted after the 70% due of asm1

• Midterm time update:
  • Previous: Tue of week 5 (Feb 6)
  • Now: Tue of week 6 (Feb 13) during lecture time
Example:

- What is the value of `argc` if user entered this command to run the program?

```
./a.out -r 5 -c 4
```

- What does the 2-d array (argv) look like for the above command-line arguments?
Lecture Topics:

• Complexity Analysis
How to compare Data Structures?

• We have different data structures, how to compare them?

• We want a way to characterize runtime or memory usage that is completely platform-independent
  • i.e. does not depend on hardware, operating system, programming language, etc.
Complexity Analysis

• Use Complexity Analysis to help make platform-independent comparisons of data structures
  • Also known as Big O

• To do this, we describe how a data structure’s runtime or memory usage changes relative to a change in the input size ($n$)
  • Importantly, we want to describe how data structures behave in the limit, as $n$ approaches $\infty$ (infinity)
Big O

• We use Big O notation to assess a data structure or algorithm’s performance.

• Big O notation: a tool for characterizing a function in terms of its growth rate
  • Indicate an upper bound on the function’s growth rate, known as growth order
**Big O**

$g(x)$ provides an upper bound on $f(x)$

For $x > x_0$,

$g(x) > f(x)$

$g(x)$ is $O(f(x))$
Common growth order functions

- \( n! \)
- \( 2^n \)
- \( n^2 \)
- \( n \log_2 n \)
- \( n \)
Common growth order functions

- $O(1)$ – constant complexity
- $O(\log n)$ – log-n complexity
- $O(\sqrt{n})$ – root-n complexity
- $O(n)$ – linear complexity
- $O(n \log n)$ – n-log-n complexity
- $O(n^2)$ – quadratic complexity
- $O(n^3)$ – cubic complexity
- $O(2^n)$ – exponential complexity
- $O(n!)$ – factorial complexity
Compute Runtime Complexity

**Input size** \( n \)

```java
int sum = 0;
for (int i = 0; i < n; i++) {
    sum += array[i];
}
return sum;
```

• The instruction `int sum = 0;` executes in some constant time \( c_1 \) independent of \( n \)

• Each iteration of the loop executes in some constant time \( c_2 \), and this happens \( n \) times

• The return statement executes in some constant time \( c_3 \) independent of \( n \)

• So runtime is \( c_1 + c_2 \times n + c_3 \)

• \( c_1, c_2, \) and \( c_3 \) depend on the particular computer running this function, so we ignore them to figure out run-time complexity

• Thus, this function grows on the order of \( n \), a.k.a. its run-time complexity is \( \mathcal{O}(n) \)
Compute Runtime Complexity

input size: $n \rightarrow \# of nodes$

```c
struct node* push (struct node * head, int val) {
    struct node *temp = new node;
    temp->val = val;
    temp->next = head;
    head = temp;
    return head;
}
```

- Every instruction in this function executes in some constant time, independent of $n$
- Thus we ignore them to figure out runtime complexity.
- Complexity: $O(c1+c2+c3+c4+c5) = O(1)$
More examples

• Loops are one of the main determinants of a program’s complexity

• for (int i = 0; i < n; i++) {
  ...
  \( O(1) \)
  ...
} \( O(n) \)

• for (int i = n; i > 0; i/=2) {
  ...
} \( 2^i = n \) \( i = \log_{2} n \) \( O(\log n) \)

• for (int i = 0; i*i < n; i++) {
  ...
  \( i^2 < n \)
  \( i < \sqrt{n} \) \( O(\sqrt{n}) \)
More examples

- for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    ...
  }
}

- for (int i = n; i > 0; i/=2) {
  for (int j = 0; j < n; j++) {
    ...
  }
}

\[ \text{inner loop} \]

\[ \text{\( n \times n = n^2 \)} \]

\[ \text{n terms} \]

\[ \text{\( \Omega(n^2) \)} \]

\[ \log n \]

\[ O(\log n) \]
Determining a program’s complexity

```c
void bubble_sort(struct node *head, int size) {
    ...
    for (int j = 1; j < size; j++) {
        for (int i = 0; i < size-j; i++) {
            if (current->val > current->next->val) {
                //swap O(1)
                //move current to next node O(1)
            }
            current = head;  O(1)
        }
    }
}
```

- Number of iterations:
  - $O((n - 1) + (n - 2) + (n - 3) + ... + 2 + 1)$
  - $= O \left( \frac{(n-1+1) \cdot (n-1)}{2} \right)$
  - $= O \left( \frac{n^2 - n}{2} \right)$
  - $= O(n^2 - n)$ Is this the final answer?

\[
\begin{align*}
    \text{# of iter} & \quad j \\
    1 & \quad n - 1 \\
    2 & \quad n - 2 \\
    3 & \quad n - 3 \\
    \vdots & \quad \vdots \\
    n - 3 & \quad n - (n - 3) \\
    n - 2 & \quad \vdots \\
    n - 1 & \quad \vdots \\
\end{align*}
\]
Dominant components

• When a growth order function has additive terms, one of those will dominate the others
  • Specifically, function $f(n)$ dominates $g(n)$ if $n_0:n > n_0$, $f(n) > g(n)$

• In these cases, we simply ignore the non-dominant terms
  • i.e. $n^2$ — $n$, $n^2$ dominates $n$, so we ignore $n$, and we say this complexity is $O(n^2)$
Dominant components

• Example: If an algorithm grows on the order of \( n^2 + n + \log n + 1 \), what is the complexity of the algorithm using big O notation?

\[ O(n^2) \]

• Takeaway: When loops are executed in sequence, the loop with the highest runtime complexity will determine the overall runtime complexity of the whole function.
Dominant components

• Example: What’s the runtime complexity of the following?

```c
for (i = 0; i < n; i++) {
    ...
}

for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        ...
    }
}

for (i = 0; i < n; i++) {
    ...
}
```

\[ O(n^2) \]
Calculating time from Big O

If a \(O(n)\) algorithm takes 32ms to sum 10,000 elements, how long will it take to sum 20,000?

\[
\frac{f(n_1)}{f(n_2)} = \frac{t_1}{t_2} \Rightarrow \frac{n_1}{n_2} = \frac{t_1}{t_2} \Rightarrow \frac{10000}{20000} = \frac{32}{t_2} \Rightarrow t_2 = \frac{32}{\frac{10000}{20000}} = 64\text{ms}
\]

For an \(O(n)\) algorithm, if size doubles, execution time doubles.

What if this algorithm has \(O(n^2)\) complexity?

\[
\frac{f(n_1)}{f(n_2)} = \frac{t_1}{t_2} \Rightarrow \frac{n_1^2}{n_2^2} = \frac{t_1}{t_2} \Rightarrow \frac{n_1^2}{(2n_1)^2} = \frac{t_1}{t_2} \Rightarrow \frac{1}{4} \cdot \frac{n_1^2}{n_1^2} = \frac{t_1}{t_2}
\]

\[
t_2 = 4 \times t_1 = 4 \times 32 = 128\text{ms}
\]

Runtime goes up by a factor of 4.
Calculating time from Big O

• Ex. Merge sort, which is an $O(n \log n)$ algorithm, takes 96ms to sort an array of size 4000. Given this result, approximately how long merge sort will take to sort an array of size 1,000,000?

$T(n) \approx \frac{96 \text{ms}}{2^{12}} \cdot 2^{20}$

• Hint: 4000 ≈ $2^{12}$, 1,000,000 ≈ $2^{20}$
Worst case, Best case, and avg. case

• Note that the worst case, best case, and average case complexities of a data structure or an algorithm can differ, for example:

```c
int linear_search(int q, int* array, int n) {
    for (int i = 0; i < n; i++) {
        if (array[i] == q) {
            return i;
        }
    }
    return -1;
}
```

• Worst case: $O(n)$: if $q$ appears to be the last element / does not exist
• Best case: $O(1)$: if $q$ appears to be the first element
• Avg. case: $O(n)$: run about $n/2$ iterations, drop $\frac{1}{2}$
Real-world Consideration

• Your program will only perform as well as your design
  • Constant factors can still play a part

• Suppose you have two data structure or algorithms perform the same task:
  • A) $1,000,000n \rightarrow O(n)$
  • B) $2n^2 \rightarrow O(n^2)$
  • Which one is better?
    • It depends
Complexity of dynamic array insertion

• Recall: dynamic array insertion
  • Case 1: if size < capacity
    • Insert the new element
  • Case 2: if size ≥ capacity
    • Step 1: allocate a new array that has twice the capacity
    • Step 2: copy all elements from data to new array
    • Step 3: delete the old data array and update data pointer
    • Step 4: Insert the new element

• Group Activity: What is the best-case, worst-case, and average case runtime complexities? Using Big O