Big-O

Introduction

Big-O is a technique that lets you quickly understand an algo's efficiency

- Big-O of n^2: if you pass in n items to it, it will take roughly n^2 steps to process its data
- **NOT EXACT**, just rough estimates

Ways to Measure Speed of Algorithm

Time it takes for an algorithm to run

1. Runtime?
    - Nah, not that useful
2. Number of computer instructions it takes to solve a problem of a given size
    - Also not too much info
    - An algo might look efficient when applied to a small amount of data (e.g. 1000 numbers) but really slow when applied to lot of data

> Understand how algo performs under all circumstances

3. The number of instructions used by an algorithm as a function of the size of the input data

- Sort N numbers
    - Algorithm A takes $5n^2$ instructions
    - B takes $37000n$ instructions
    - Compare
        - If sort 1000 numbers
            - A takes 5M instructions, B takes 37M
        - If sort 10000 numbers
            - A takes 500M instructions, B takes 370M

Concept

> Big-O approahc measures an algorithm by the **gross number of steps** that it requires to **process an input of size N** in the *WORST CASE SCENARIO*

- Ignore coefficients and lower-order terms
    - Algorithm X requires $5n^2 + 3n + 20$ steps -> Big-O of Algorithm X is N^2

Compute Big-O

1. Determine number of operations an algorithm performs

- `f(n)`
- Operations are any of the following:
  - Access an item (e.g. in the array)
  - Evaluate a mathematical expression
  - Traverse a single link in a linked list

Example: Compute `f(n)`: the # of critical operations that this algorithm performs

```
int arr[n][n];

for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        arr[i][j] = 0;
    }
}
```

1. Algorithm initializes the value of `i` once: `f(n) += 1`
2. Performs `n` comparisons between `i` and `n`: `f(n) += n`
3. Increments the variabl `i`, `n` times: `f(n) += n`
4. Initializes the value of `j`, `n` different times: `f(n) += n`
5. Performs n^2 comparisons between `j` and `n`: `f(n) +=` n^2
6. Increments the variable `j`, n^2 times: `f(n) +=` n^2
7. Algorithm sets `arr[i][j]`'s value n^2 times: `f(n) +=` n^2

$f(n) = 1 + n + n + n + n^2 + n^2 + n^2 = 3n^2 + 3n + 1$

2. Keep the most significant term of that function and throw away the rest

$f(n) = 3n^2 + 3n + 1$ BECOMES $f(n) = 3n^2$

3. Remove any constant multiplier from the function

$f(n) = 3n^2$ BECOMES $f(n) = n^2$

4. BIG-O

$f(n) = 3n^2 + 3n + 1$ is **$O(n^2)$**

Simplified Approach

There's no need to compute exact `f(n)` because we end up throwing away all lower order terms and coefficients

JUST focus on most frequently occuring operators

```
arr[n][n] = 0;
for(int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        arr[i][j] = 0;
```

Just look at...

```
arr[i][j] = 0;
```

$f(n) = n^2$

Hence the algorithm is $O(n^2)$

- To process n items, this algorithm requires roughly n^2 operations

Order of Big-O Complexity

> $log2n$ < N < $nlog2n$ < n^2 < n^3 < 2^n

Choose the Correct Algorithm Case by Case

- If operates on a large number of items -> **Evaluate Big-O**
- If small: choose the easiest to write program is more efficient

---

Find the Big-O Challenge

```
for (int i = 0; i < n; i+=2)
    sum++;
```

- The loop runs n/2 times, ignore 1/2
- **$O(n)$**

```
k = n;
while (k > 1)
```

```
{
    sum++;
    k = k/2;
}
```

- k goes from n to n/2 to n/4 to n/8 all the way down to 1
- Since we divide by 2 each time, it takes $log2(n)$ steps to finish
- **$O(log2(n))$**

```
for (int i = 0; i < n; i++)
{
    int k = n;
    while (k > 1)
    {
        sum++;
        k = k/3;
    }
}
```

- Outer loop runs exactly n times
- Each time, the inner loop runs log3n iterators
- So the innter most code runs nlog3(n) times
- **$O(nlog3n)$**

```
void foo()
{
    int i, sum = 0;
    for (i = 0; i < n*n; i++)
        sum += i;
    for (i = 0; i < n*n*n; i++)
        sum += i;
}
```

- First loop runs exactly n^2 times
- Second loop runs exactly n^3 times
- So the overall code runs n^2 + n^3 times
- **$O(n^3)$**

Address Complicated Situations

Step 1: Locate all loops that **don't run for a fixed number of iterations** and determine the **maximum** number of iterations each loop could run for

Step 3: Turn these loops into loops with a fixed # of iterations, using their **maximum possible iteration count**

```
for(int i = 0; i < n; i++)
    for (int j = 0; j < i; j++)
        cout << j;
```

- The inner loop runs a variable number of iterations depending on `i`'s values
- The maximum value that `i` can take is `n-1`
- Round that up to `n` and make inner loop a fixed loop

```
for(int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        cout << j;
```

- **$O(n^2)$**

```
for (int i = 0; i < n; i++)
{
    Circ arr[n];
    arr[i].setRadius(i);
}
```

- Each time the outer loop runs once, the Circ constructor runs `n` times to initializes the array of size n
- **$O(n^2)$**

```
for (int i = 0; i < n; i++)
{
    int k = i;
    while (k > 1)
    {
        sum++;
        k = k/2;
    }
}
```

- `i`'s largest value is going to be n
- replace `i` with `n` inside

- **$O(nlog2(n))$**

Multi-Input Algorithms

- Must take into account both independent sizes because either variable could dominate the other

```
void buffet(string people[], int p, string foods[], int f)
{
    int i, j;
    for (i = 0; i < p; i++)
        for (j = 0; j < f; j++)
            cout << peope[i] << "ate "
                << foods[j] << endl;
}
```

- **$O(p*f)$**

```
void tinder(string csmajors[], int c, string eemajors[], int e)
{
    for (int i = 0; i < c; i++)
        for (int j = 0; j < c; j++)
            cout ...
    for (int k = 0; k < e; k++)
        cout << eemajors[k] << ...
}
```

- **$O(c^2 + e)$**

Do not forget to elimiate lower-order terms for each independent variable

```
void tinder(string csmajors[], int c, string eemajors[], int e)
{
    for (int i = 0; i < c; i++)
        for (int j = 0; j < c; j++)
            cout ...
    for (int m = 0; m < c; m++) //c iterations, ignore
        ...

    for (int k = 0; k < e; k++)
        cout << eemajors[k] << ...
}
```

- **$O(c^2 + e)$**

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

```
    if (i == n/2)
    {
        for (int k = 0; k < q; k++)
            cout << "Muahahahaha!";
    }
    else
        cout << "Brrp!";
}
```

- We perform `n` outer iterations
- The one time when `i` is equal to `n/2`
    - We run the inner loop which has `q` iterations
- So n iterations plus a one-time run of q iterations: **$O(n + q)$**

The STL and Big-O

Example:

If we write a loop of our own that runs D times, and each iteration of our loop searches for an item in a set holding n items

```
void inDict(set<string> &d, string w)
{
    if (d.find(w) == d.end())
        cout << w << " isn't in dictionary!";
}

void spellCheck(set<string> &dict, string doc[], int D)
{
    for (int i = 0; i < D; i++)
        inDict(dict, doc[i]);
}
```

- To search a set of n items for a single value requires `log2(n)` steps
- Repeat this search operation D different times
- **$O(D*log2(n))$**

```
void printNums(vector<int> &v)
{
    int q = v.size();
    for (int i = 0; i < q; i++)
    {
        int a = v[0];
        cout << a;
```

```
        v.erase(v.begin());
        v.push_back(a);
    }
}
```

- Loop runs q times
- Accessing an element is O(1)
- Erase the first element is O(q)
- Push back an element is O(1)
- **$O(q^2)$**

```
int main()
{
    set<int> nums;
    for (int i=0; i < q; i++)
        nums.insert(i);
}
```

- inserting a number into a set with n items take log2n steps
- but our set starts out empty, there are 0 items
- the set will have a different number of items during each iteration of the loop

> When evaluating STL-based algorithms, first determine the maximum number of items each container could possibly hold

- Then do Big-O analysis under the assumption that each container always holds exactly this number of items
- Assume that our set always has q items in it -> each time we insert an item into our set is log2(q) steps
- If loop runs a total of q iterations and each iteration insertion costs log2(q)
- Total cost: **$O(q\log_2(q))$**

```
//assume p items in vector v
void clearFromFront(vector<int> &v)
{
    while(v.size() > 0)
    {
        v.erase(v.begin()); //erase 1st item
    }
}
```

- Assume vector has p items, and we delete one item per iteration, our loop runs for p steps
- Deletion of first item costs p steps, assuming vector always has p items no matter what
- **$O(p^2)$**

```
//assume s starts out empty
void addItems(set<int> &s, int q)
{
    for (int i = 0; i < q*q; i++)
    {
        s.insert(i);
    }
}
```

- For loop runs a total of q^2 iterations
- To compute cost of an STL operation, assume the set always holds its max # of items
    - Set here will eventually hold q^2 items, so insert a new item costs $log2(q^2)$
- **$O(q^2log2(q^2))$**

picture 1

Space Complexity

Space complexity is the big-o of how much storage your algorithm uses, as a function of the data input size, `n`

Example

```
void reverse(int array[], int n)
{
    int tmp, i;
    for (i = 0; i < n/2; ++i)
    {
        tmp = array[i];
        array[i] = array[n-i-1];
        array[n-i-1] = tmp;
    }
}
```

- This uses just 2 new variables `tmp` and `i` no matter how big the array is
- Space Complexity: **O(1)** or **O(Constant)**

```
void reverse(int array[], int n)
{
```

```
    int *tmparr = new int[n];
    for (int i = 0; i < n; ++i)
        tmparr[n-i-1] = array[i];
    ...
}
```

- Uses a new array of size n to process the input array of size n
- Space Complexity: **O(n)**

Space Complexity and Recursion

Without Recursion

```
void printNums(int n)
{
    int i;
    for (i = nl i >= 0; i--)
        cout << i << "\n";
}
```

- Uses a 4 byte memory slot no matter how large n is
- **O(1)**

With Recursion

```
void printNums(int n)
{
    if (n < 0) return;
    cout << n << "\n";
    printNums(n-1);
}
```

- Creates a new variable for each of the n levels of recursion
- **O(n)**