Data Structures & Algorithms

Big O Notation

B.Bhuvaneswaran, AP (SG) / CSE

9791519152

bhuvaneswaran@rajalakshmi.edu.in

RAJALAKSHMI ENGINEERING COLLEGE

An AUTONOMOUS Institution

Affiliated to ANNA UNIVERSITY, Chennai

The Idea Behind Big O Notation

- Big O notation is the language we use for talking about how long an algorithm takes to run.
- It's how we compare the efficiency of different approaches to a problem.
- With big O notation we express the runtime in terms of - *how quickly it grows relative to the input, as the input gets arbitrarily large*.

How Quickly the Runtime Grows

- It's hard to pin down the *exact runtime* of an algorithm.
- It depends on the speed of the processor, what else the computer is running, etc.
- So instead of talking about the runtime directly, we use big O notation to talk about *how quickly the runtime grows*.

Relative to the Input

- If we were measuring our runtime directly, we could express our speed in seconds.
- Since we're measuring *how quickly our runtime grows*, we need to express our speed in terms of...something else.
- With Big O notation, we use the size of the input, which we call "*n*".
- So we can say things like the runtime grows "on the order of the size of the input "*O(n)*" or on the order of the square of the size of the input "$O(n^2)$".

As the Input Gets Arbitrarily Large

- Our algorithm may have steps that seem expensive when *n* is small but are eclipsed eventually by other steps as *n* gets huge.
- For big O analysis, we care most about the stuff that grows fastest as the input grows, because everything else is quickly eclipsed as gets very large (If you know what an asymptote is, you might see why "big O analysis" is sometimes called "asymptotic analysis").

Examples

```
void printFirstItem(const int *items)
{
        printf("%d\n", items[0]);
}
```

Examples

- This function runs in *O(1)* time (or "constant time") relative to its input.
- The input array could be 1 item or 1,000 items, but this function would still just require one "step".

Examples

```
void printAllItems(const int *items, size_t size)
{
        size_t i;
        for (i = 0; i < size; i++)
        {
                printf("%d\n", items[i]);
        }
}
```

Examples

- This function runs in *O(n)* time (or "linear time"), where *n* is the number of items in the array.
- If the array has 10 items, we have to print 10 times.
- If it has 1,000 items, we have to print 1,000 times.

Examples

```
void printAllPossibleOrderedPairs(const int *items, size_t size)
{
        size_t i, j;
        for (i = 0; i < size; i++)
        {
                for (j = 0; j < size; j++)
                {
                        printf("%d, %d\n", items[i], items[j]);
                }
        }
}
```

Examples

- Here we're nesting two loops. If our array has *n* items, our outer loop runs *n* times and our inner loop runs *n times for each iteration of the outer loop*, giving us n^2 total prints.
- Thus this function runs in $O(n^2)$ time (or "quadratic time").
- If the array has 10 items, we have to print 100 times.
- If it has 1,000 items, we have to print 10, 00,000 times.

N Could be the Actual Input, or the Size of the Input

- Both of these functions have *O(n)* runtime, even though one takes an integer as its input and the other takes an array:

```
void sayHiNTimes(size_t n)
{
   size_t i;
   for (i = 0; i < n; i++)
   {
      printf("hi\n");
   }
}
```

```
void printAllItems(const int *items, size_t size)
{
   size_t i;
   for (i = 0; i < size; i++)
   {
      printf("%d\n", items[i]);
   }
}
```

- So sometimes *n* is an *actual number* that's an input to our function, and other times *n* is the *number of items* in an input array (or an input map, or an input object, etc.).

Drop the Constants

- This is why big O notation *rules*.
- When you're calculating the big O complexity of something, you just throw out the constants.

Drop the Constants

```
void printAllItemsTwice(const int *items, size_t size)
{
        size_t i;
        for (i = 0; i < size; i++)
        {
                printf("%d\n", items[i]);
        }
        // once more, with feeling
        for (i = 0; i < size; i++)
        {
                printf("%d\n", items[i]);
        }
}
```

- This is *O(2n)*, which we just call *O(n)*.

Drop the Constants

```
void printFirstItemThenFirstHalfThenSayHi100Times(const int *items, size_t size)
{
        size_t i, middleIndex, index;
        printf("%d\n", items[0]);
        middleIndex = size / 2;
        index = 0;
        while (index < middleIndex)
        {
                printf("%d\n", items[index]);
                index++;
        }
        for (i = 0; i < 100; i++)
        {
                printf("hi\n");
        }
}
```

- This is *O(1 + n/2 + 100)*, which we just call *O(n)*.

Why can we get away with this?

- Remember, for big O notation we're looking at what happens as *n* gets arbitrarily large.
- As *n* gets really big, adding 100 or dividing by 2 has a decreasingly significant effect.

Drop the Less Significant Terms

```
void printAllNumbersThenAllPairSums(const int *numbers, size_t size)
{
        size_t i, j;
        printf("these are the numbers:\n");
        for (i = 0; i < size; i++)
        {
                printf("%d\n", numbers[i]);
        }
        printf("and these are their sums:\n");
        for (i = 0; i < size; i++)
        {
                for (j = 0; j < size; j++)
                {
                        printf("%d\n", numbers[i] + numbers[j]);
                }
        }
}
```

Drop the Less Significant Terms

- Here our runtime is $O(n + n^2)$, which we just call $O(n^2)$.
- Even if it was $O(n^2/2 + 100n)$, it would still be $O(n^2)$.
- Similarly:
  - $O(n^3 + 50n^2 + 10000)$ is $O(n^3)$
  - $O((n + 30) * (n + 5))$ is $O(n^2)$
- Again, we can get away with this because the less significant terms quickly become, well, less significant as *n* gets big.

We're Usually Talking About the "worst case"

- Often this "worst case" stipulation is implied. But sometimes you can impress your interviewer by saying it explicitly.
- Sometimes the worst case runtime is significantly worse than the best case runtime:

We're Usually Talking About the "worst case"

```
int contains(const int *haystack, size_t size, int needle)
{
        size_t i;
        // does the haystack contain the needle?
        for (i = 0; i < size; i++)
        {
                if (haystack[i] == needle)
                {
                        return 1;
                }
        }
        return 0;
}
```

We're Usually Talking About the "worst case"

- Here we might have 100 items in our haystack, but the first item might be the needle, in which case we would return in just 1 iteration of our loop.
- In general we'd say this is *O(n)* runtime and the "worst case" part would be implied.
- But to be more specific we could say this is worst case *O(n)* and best case *O(1)* runtime.
- For some algorithms we can also make rigorous statements about the "average case" runtime.

Space Complexity: the Final Frontier

- Sometimes we want to optimize for using less memory instead of (or in addition to) using less time.
- Talking about memory cost (or "space complexity") is very similar to talking about time cost.
- We simply look at the total size (relative to the size of the input) of any new variables we're allocating.

Space Complexity: the Final Frontier

- This function takes *O(1)* space (we use a fixed number of variables):

```
void sayHiNTimes(size_t n)
{
        size_t i;
        for (i = 0; i < n; i++)
        {
                printf("hi\n");
        }
}
```

Space Complexity: the Final Frontier

- This function takes *O(n)* space (the size of hiArray scales with the size of the input):

```
char** arrayOfHiNTimes(size_t n)
{
        size_t i;
        char **hiArray = calloc(n, sizeof(char*));
        assert(hiArray != NULL);
        for (i = 0; i < n; i++)
        {
                hiArray[i] = strdup("hi");
                assert(hiArray[i] != NULL);
        }
        return hiArray;
}
```

Space Complexity: the Final Frontier

- Usually when we talk about space complexity, we're talking about *additional space*, so we don't include space taken up by the inputs.
- For example, this function takes constant space even though the input has *n* items:

Space Complexity: the Final Frontier

```
int getLargestItem(const int *items, size_t size)
{
        size_t i;
        int largest = INT_MIN;
        for (i = 0; i < size; i++)
        {
                int item = items[i];
                if (item > largest)
                {
                        largest = item;
                }
        }
        return largest;
}
```

Space Complexity: the Final Frontier

- Sometimes there's a tradeoff between saving time and saving space, so you have to decide which one you're optimizing for.

Queries?

Thank You…!