Runtime Analysis

1 Introduction

The (worst case) runtime of an algorithm is the function $T$ from the natural numbers to the natural numbers defined by

$$T(n) := \text{maximum number of primitive computational steps the algorithm executes on any input instance of size } n.$$  

Primitive steps are low-level instructions with execution time that depends on hardware/software environment, but is nevertheless constant.

- Assigning a value to a variable
- Arithmetic operations ($+, -, *, /$)
- Comparison of integers
- Single memory read or write

Non-primitive steps are those which are composed of many individual steps:

- Loops
- Subroutines

Note: The instruction $\text{pow}(x, n)$ which computes $x^n$ is not a primitive step. It is a subroutine which multiplies $x$ with itself $n$ times. Therefore the runtime is $O(n)$.

There are a few key points about the definition you need to remember.

- It is a function. We care about the efficiency of the algorithm as the input sizes get larger.
- It measures the number of steps in the worst case , i.e., the input of size $n$ on which the algorithm performs the worst.
- Primitive steps are instructions for which the amount of “work” does not depend on the input.
2 Some Examples

Example 1. What is the runtime, as a function of $n$, of the following loop?
for ($i = 1; i <= n; i + +$)
print “howdy”

Solution. Notice that the instruction print “howdy” is composed of some constant number $c$ of steps (depending on the language used, the compiler, the CPU architecture, etc.), because it does not depend on $n$. Each iteration of the loop therefore performs $c$ steps. In summation form,
$$\sum_{i=1}^{n} c$$
which is equal to $cn$. Thus the runtime of the code is $O(n)$.

Example 2. What is the run time, as a function of $n$, of the following loop?
for ($i = 1; i <= n; i + +$)
for ($j = 1; j <= n; j + +$)
costant number of operations

Solution. In each iteration of the inner loop a constant number of operations are performed. Denote this constant by $c_1$. Therefore the runtime of the inner loops is
$$\sum_{j=1}^{n} c_1$$
which is equal to $c_1n$. During each iteration of the outer loop, the algorithm executes a constant number of operations (call this $c_2$) plus the number of operations performed by the inner loop. Therefore, the time taken by each iteration of the outer loops is
$$c_2 + \sum_{j=1}^{n} c_1$$
Thus, the runtime of the above code is
$$\sum_{i=1}^{n} \left( c_2 + \sum_{j=1}^{n} c_1 \right) = \sum_{i=1}^{n} c_2 + \sum_{i=1}^{n} \sum_{j=1}^{n} c_1$$
$$= c_2n + \sum_{i=1}^{n} c_1n$$
$$= c_2n + c_1n^2.$$  
Hence, the runtime of the code is $O(n^2)$.

Example 3. What is the runtime, as a function of $n$, of the following loop?
for ($i = 1; i <= n; i + +$)
for ($j = 1; j <= i; j + +$)
costant number of operations

Solution. In each iteration of the inner loop a constant number of operations are performed. Denote this constant by $c_1$. Therefore the runtime of the inner loops is
$$\sum_{j=1}^{i} c_1$$
which is equal to $c_1 n$. During each iteration of the outer loop, the algorithm executes a constant number of operations (call this $c_2$) plus the number of operations performed by the inner loop. Therefore, the time taken by each iteration of the outer loops is

$$c_2 + \sum_{j=1}^{i} c_1$$

Thus, the runtime of the above code is

$$\sum_{i=1}^{n} \left( c_2 + \sum_{j=1}^{i} c_1 \right) = \sum_{i=1}^{n} c_2 + \sum_{i=1}^{n} \sum_{j=1}^{i} c_1$$

$$= c_2 n + \sum_{i=1}^{n} c_1 i$$

$$= c_2 n + c_1 \sum_{i=1}^{n} i$$

$$= c_2 n + c_1 n(n + 1)/2$$

$$= c_2 n + c_1 n^2/2 + c_1 n/2.$$

Hence, the runtime of the code is $O(n^2)$.

### 3 Big Oh and Runtime

You’ve noticed that in the previous examples, we would give the big Oh of the runtime, instead of the actual function $T$. In this class, we will typically only give the big Oh of the runtime of an algorithm. At first, this seems like we are needlessly giving away information. However, there are several reasons to do this.

- We want our measure of runtime to be as general as possible.
  - We want to analyze algorithms, not implementations.
  - The constants in the above examples were machine, language and implementation dependent.

- We care about how the efficiency grows with the input size.
  - Exact values for constants become less important as the input size grows.

- We do not lose much by using big Oh.
  - Works well in practice.

- Big Oh makes analyzing the runtime of algorithms easier to calculate.
  - Getting the exact number of steps would be tedious and difficult.
4 More Examples

Consider the following algorithm, which, given an array $a$, returns 0 if the first element is 0, and otherwise returns the largest integer contained in it. What is the runtime of largestInt in big Oh notation, as a function of $n$, the size of the array?

**Example 4.** largestInt(array $a$)

```cpp
    if($a[0] == 0$) return 0
    int max = $a[0]$
    for ($i = 0; i < n; i++$)
        if(max $<$ $a[i]$) set max = $a[i]$
    return max
```

The answer is $O(n)$. A couple points to notice:

- Although the worst case is $O(n)$, in the best case, this algorithm takes constant time.
- Besides that, this is essentially the analysis as Example 1.

**Example 5.** What is the runtime, as a function of $n$, of the following loop?

```cpp
    for ($i = n; i >= 1; i = i/2$)
        constant number of operations
```

**Solution.** The number of steps is equal to

$$\text{(number of steps during each iteration)} \times \text{(number of iterations)}.$$ 

During each iteration, some constant $c$ number of operations are executed. To complete the analysis, we need to calculate the number of iterations (as a function of $n$). Notice that the value of $i$ (approximately) changes as per the following progression

$$n, n/2, n/4, \ldots, 1.$$ 

We see that, after the $k$th iteration, the value of $i$ is (approximately) $n/2^k$. The number of iterations is the value of $k$ when we exit the loop. Since we exit the loop when $i = 1$, we can find the value of $k$ when we exit the loop by solving $n/2^k = 1$. By taking the logarithm, we see that $k = \log n$. Since we were using approximation, the number of iterations is $O(\log n)$. Finally, because the number of operations executed during each iteration is $c$, we see that the runtime of the loop is $O(\log n)$. 

**Remark:** Technically, in the above solution, we are doing integer division, so the value of $i$ is only approximately equal to $n/2, n/4, \ldots$. However, there is not much difference between the actual value of $i$ and our approximations. In particular, the number of iterations calculated is within 1 of the actual number of iterations. It might be helpful to test this yourself. For example, when $n = 30$, the value of $i$ progresses 100, 50, 25, 12, 6, 3, 1. If you take a calculator, you see that $\log 100$ is 6.64\ldots.