Runtime Analysis

May 29, 2018
Topics

- What is algorithm analysis
- Big O, big Ω, big Θ notations
- Examples of algorithm runtimes
Involved in many important runtime results: sorting, binary search, etc.

Logarithms grow slowly, much more slowly than any polynomial but faster than a constant

Definition: $\log_b n = k$ if $b^k = n$

- $b$ is the base of the log

Examples:

- $\log_2 8 = 3$ because $2^3 = 8$
- $\log_{10} 100 = 2$ because $10^2 = 100$
- $2^{10} = 1024$ (1K), so $\log_2 1024 = 10$
- $2^{20} = 1M$, so $\log 1M = 20$
- $2^{30} = 1G$, so $\log 1G = 30$
Some Useful Identities of Logarithm

- \( \log(nm) = \log(n) + \log(m) \)
- \( \log(n/m) = \log(n) - \log(m) \)
- \( \log(n^k) = k \log(n) \)
- \( \log_a(b) = \frac{\log b}{\log a} \)
- If the base of log is not specified, assume it is base 2
  - log: base 2
  - ln: base e
Logarithms

- It requires $\log_k n$ digits to represent $n$ numbers in base $k$.
- It requires approximately $\log_2 n$ multiplications by 2 to go from 1 to $n$.
- It requires approximately $\log_2 n$ divisions by 2 to go from $n$ to 1.
- Computers work in binary, so in order to calculate how many numbers a certain amount of memory can represent we use $\log_2$. 

16 bits of memory can represent $2^{16}$ different numbers, $2^{10+6} = 2^{10} \times 2^{6} = 64K$

32 bits of memory can represent $2^{32}$ different numbers, $2^{30+2} = 2^{30} \times 2^{2} = 4G$ – see previous slide

64 bits (most of today’s computers address space)
An algorithm is a clearly specified set of instructions the computer will follow to solve a problem.

When we develop an algorithm we want to know how many resources it requires.

We try to develop an algorithm to use as few resources as possible.

Let $T$ and $n$ be positive numbers. $n$ is the size of the problem and $T$ measures a resource: Runtime, CPU cycles, disk space, memory etc.

Order of growth can be important. For example – sorting algorithms can perform quadratically or as $n \times \log(n)$. Very big difference for large inputs.
Algorithm Analysis

- Resources: space and time
- Common functions used in runtime analysis
  - $1$, constant
  - $\log n$, logarithmic
  - $n$, linear
  - $n \log n$, superlinear
  - $n^2$, quadratic
  - $n^3$, cubic
  - $2^n$, exponential
  - $n!$, factorial

$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$
Motivation for Big O

- $F(n) = 0.0001n^3 + 0.001n^2 + 0.01$ versus $G(n) = 1000n$
- It doesn’t make sense to say $F(n) < G(n)$
- For *sufficiently large* $n$, the value of a function is largely determined by the *dominant* term
- When $n$ is small, we just don’t care that much about runtime
- Big-Oh notation is used to capture the most dominant term in a function.
Big O Definition

- \( T(n) \) is \( O(F(n)) \) if there are positive constants \( c \) and \( N_0 \) such that \( T(n) \leq c \cdot F(n) \), for all \( n \geq N_0 \)
- \( T(n) \) is bounded by a multiple of \( F(n) \) from above for every big enough \( n \)
- \( F(n) \) is an upper bound of \( T(n) \)
- Example: Show that \( 2n + 4 = O(n) \)
- Example: Show that \( 2n + 4 = O(n^2) \)
Example

- \(2n + 4 = O(n)\)
- To solve this, you have to actually give two constants, \(c\) and \(N_0\) such that \(2n + 4 \leq c \cdot n\) for every \(n \geq N_0\)
- For example, we can pick \(c = 4\) and \(N_0 = 2\)
Big $\Omega$ Definition

- $T(n)$ is $\Omega(F(n))$ if there are positive constants $c$ and $N_0$ such that $T(n) \geq c \cdot F(n)$, for all $n \geq N_0$
- $T(n)$ is bounded by a multiple of $F(n)$ from below for every big enough $N$
- $F(n)$ is a lower bound of $T(n)$
- Example: Show that $2n + 4 = \Omega(n)$
- Example: Show that $2n + 4 = \Omega(\log n)$
Examples

- $3n^2 - 100n + 6 = O(n^2)$
- $3n^2 - 100n + 6 = O(n^3)$
- $3n^2 - 100n + 6 \neq O(n)$
- $3n^2 - 100n + 6 = \Omega(n^2)$
- $3n^2 - 100n + 6 \neq \Omega(n^3)$
- $3n^2 - 100n + 6 = \Omega(n)$
Big $\Theta$ Definition

- Often the upper and lower bounds are different
  - Needs further research to close the gap
- When upper and lower bounds agree (a *tight* bound), the problem is solved theoretically
- $T(n)$ is $\Theta(F(n))$ if and only if $T(n)$ is $O(F(n))$ and $T(n)$ is $\Omega(F(n))$
- $F(n)$ is both the upper and lower bounds of $T(n)$
- Example: $2n + 4 = \Theta(n)$


### Runtime Table

$f(n)$: runtime

<table>
<thead>
<tr>
<th>$n$</th>
<th>$f(n)$</th>
<th>$\lg n$</th>
<th>$n$</th>
<th>$n \lg n$</th>
<th>$n^2$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>0.003 μs</td>
<td>0.01 μs</td>
<td>0.033 μs</td>
<td>0.1 μs</td>
<td>1 μs</td>
<td>3.63 ms</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>0.004 μs</td>
<td>0.02 μs</td>
<td>0.086 μs</td>
<td>0.4 μs</td>
<td>1 ms</td>
<td>77.1 years</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>0.005 μs</td>
<td>0.03 μs</td>
<td>0.147 μs</td>
<td>0.9 μs</td>
<td>1 sec</td>
<td>8.4 × 10^{15} yrs</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>0.005 μs</td>
<td>0.04 μs</td>
<td>0.213 μs</td>
<td>1.6 μs</td>
<td>18.3 min</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>0.006 μs</td>
<td>0.05 μs</td>
<td>0.282 μs</td>
<td>2.5 μs</td>
<td>13 days</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>0.007 μs</td>
<td>0.1 μs</td>
<td>0.644 μs</td>
<td>10 μs</td>
<td>1 ms</td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td></td>
<td>0.010 μs</td>
<td>1.00 μs</td>
<td>9.966 μs</td>
<td>100 μs</td>
<td>100 ms</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td></td>
<td>0.013 μs</td>
<td>10 μs</td>
<td>130 μs</td>
<td>10 sec</td>
<td>10 sec</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td></td>
<td>0.017 μs</td>
<td>0.10 ms</td>
<td>1.67 ms</td>
<td>16.7 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td></td>
<td>0.020 μs</td>
<td>1 ms</td>
<td>19.93 ms</td>
<td>1.16 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td></td>
<td>0.023 μs</td>
<td>0.01 sec</td>
<td>0.23 sec</td>
<td>115.7 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000,000</td>
<td></td>
<td>0.027 μs</td>
<td>0.10 sec</td>
<td>2.66 sec</td>
<td>31.7 years</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000,000</td>
<td></td>
<td>0.030 μs</td>
<td>1 sec</td>
<td>29.90 sec</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Runtime Analysis

- We care less about constants, so $100N = O(N)$.
  $100N + 200 = O(N)$.
- When the runtime is estimated as a polynomial we care about the leading term only.
- Thus $3n^3 + n^2 + 2n + 17 = O(n^3)$ because eventually the leading cubic term is bigger than the rest.
- For a good estimate on the runtime it’s good to have both the $O$ and the $\Omega$ estimates (upper and lower bounds).
Big O is transitive: If \( f(n) = O(g(n)) \) and \( g(n) = O(h(n)) \), then \( f(n) = O(h(n)) \)

**Rule for sums** (two consecutive blocks of code)
- If \( T_1(n) = O(F(n)) \) and \( T_2(n) = O(G(n)) \)
- then \( T_1 + T_2 = O(\max(F(n), G(n))) \)

**Rule for products** (an inner loop run by an outer loop)
- If \( T_1(n) = O(F(n)) \) and \( T_2(n) = O(G(n)) \)
- then \( T_1 \cdot T_2 = O(F(n) \cdot G(n)) \)

Example: \((n^2 + 2n + 17) \cdot (2n^2 + n + 17) = O(n^2 \cdot n^2) = O(n^4)\)
Approximation: When adding up a large number of terms, multiply the number of terms by the estimated size of one term

Example: Sum of $i$ from 1 to $n$
- Average size of an element: $\frac{n}{2}$
- There are $n$ terms – the sum is $O(n^2)$
- Exact solution: $\frac{n(n+1)}{2}$

Example: Sum of $i^2$ from 1 to $n$
- Average size of an element: $\frac{n^2}{2}$
- There are $n$ terms – so the sum is $O(n^3)$
- Exact solution: $\frac{n(n+1)(2n+1)}{6}$

Example: Sum of $i^3$ from 1 to $n$
- Estimate: $O(n^4)$, Exact: $\left(\frac{n(n+1)}{2}\right)^2$
Loops of Bubble Sort

- The runtime of a loop is the runtime of the statements in the loop times the number of iterations

Example: bubble sort

```c
int bubblesort(int A[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) /* n passes of loop */
        /* n-i passes of loop */
    for (j = n-1; j > i; j--)
        }
}
```
Analysis of Bubble Sort

- Work from inside out:
  - Calculate the body of inner loop (constant – an if statement and three assignments)
  - Estimate the number of passes of the inner loop: $n - i$ passes
  - Estimate the number of passes of the outer loop: $n$ passes
    - Each pass counts $n, n - 1, n - 2, \ldots, 1$.
  - Overall $1 + 2 + 3 + \cdots + n$ passes of constant operations:
    \[
    \frac{n(n+1)}{2} = O(n^2)
    \]
- This is not the fastest sorting algorithm, but it is simple and works in-place
  - Good for small size input
- We will go back to sorting later in the course
Recursive functions perform some operations and then call themselves with a different (usually smaller) input.

Example: factorial

```c
int factorial (int n) {
    if (n <= 1) return 1;
    return n * factorial(n-1);
}
```
Recursive Analysis

- Let us define $T(n)$ as a function that measures the runtime.
- $T(n)$ can be polynomial, logarithmic, exponential, etc.
- $T(n)$ may not be given explicitly in closed form, especially in recursive functions (which lend themselves easily to this kind of analysis).
- We have to find a way to derive the closed form from the recurrence formula.
Analysis of Recursive Function for Factorial

- Let us denote the runtime on input \( n \) as some function \( T(n) \) and analyze \( T(n) \).
- \( O(1) \) operations before recursive call – if statement and a multiplication.
- The recursive part calls the same function with \( n - 1 \) as input, so this part runs \( T(n - 1) \).
- So: \( T(n) = c + T(n - 1) \).
- Similarly: \( T(n - 1) = c + T(n - 2) \implies T(n) = 2c + T(n - 2) \).
- After \( n \) such equations we reach \( T(1) = k \) (just the if-statement).
- \( T(n) = (n - 1) \ast c + k = O(n) \).
- Iterative function for factorial performs the same.
A Problematic Example

The following function calculates $2^n$ for $n \geq 0$

```c
int power2(int n) {
    if (n == 0) return 1;
    return power2(n-1) + power2(n-1);
}
```

What is the problem here?
Ill-Behaved Recursion

- Each recursive call does a constant number of operations and spawns two recursive calls with $n - 1$
- $T(n) = c + 2 \times T(n - 1)$
- $T(n - 1) = c + 2 \times T(n - 2), \ldots, T(2) = c + 2 \times T(1)$
- $T(1) = k$
- $c$ is positive and therefore:
  - $T(2) > 2k$, $T(3) > 4k$, $\ldots$, $T(n) > 2^{n-1} \times k$
- $T(n)$ is exponential with $n$
- Intuitively, every call doubles the required solution time
- Bad double recursion
Ill-Behaved Recursion – Illustration

\[
T(n) = cn + T(n-1) + T(n-1) + \cdots + T(n-2) + T(n-2) + \cdots + T(n-2) + T(n-2)
\]

\[
= cn + T(n-2) + T(n-2) + \cdots + T(n-2) + T(n-2)
\]

The recursion reflects the complexity of the algorithm, with each level adding a term that depends on \( n \), leading to a cumulative effect that grows quadratically with \( n \).
Ill-Behaved Recursion

- The double recursion repeats a lot of redundant work
- The call tree looks like a big binary tree
- Double recursion is not bad, as long as the work is split too
- Example: Merge sort (good double recursion)
  - Sort recursively two halves of an array and merge
  - Call recursively twice, but on different inputs
  - The work is split between recursive calls in a smart way
- We can make power2 more efficient by calling power2(n-1) only once and multiply the result by 2
New Code

The following function calculates $2^n$ for $n \geq 0$

```c
int power2(int n) {
    if (n == 0) return 1;
    return 2*power2(n-1);
}
```

What is the runtime now?
Recurrence Formula

\[ T(n) = d \quad \text{If } n \text{ is 1} \]

\[ T(n) = 2 \times T\left(\frac{n}{2}\right) + cn \quad \text{Otherwise} \]

Notice that \( c \) and \( d \) are constants

- Identities like this come up frequently in algorithmic analysis
- It is important to have ways of solving them
- We will see a few
- One basic way is to form a recursion tree
If \( n = 2^p \) then there are \( p \) rows with \( cn \) on the right, and one last row with \( dn \) on the right.

Since \( p = \log n \), this means that the total cost is \( cn \log n + dn \).

In other words, this is what we call an \( O(n \log n) \) algorithm.
A very efficient way to hold data

The data is arranged in a binary tree structure so that every subtree rooted at element $X$ has the following properties:

- Left subtree elements are always smaller than or equal to $X$
- Right subtree elements are always larger than $X$
Binary Search Tree

- Searching the tree halves the search space at each stage
- Searching the tree is logarithmic
  - Do analysis using $T(n)$ as in previous slides
- Compare to linear search on a random array
Maximum Contiguous Subsequence Sum

- **Input:** \{-2, 11, -4, 13, -5, 2\}
- **Answer:** 20
- **Brute force:** $O(n^3)$

```java
/**
 * Cubic maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */
public static int maxSubsequenceSum(int[] a)
{
    int maxSum = 0;
    for (int i = 0; i < a.length; i++)
        for (int j = i; j < a.length; j++)
            { int thisSum = 0;
                for (int k = i; k <= j; k++)
                    thisSum += a[k];
                if (thisSum > maxSum)
                    { maxSum = thisSum;
                        seqStart = i;
                        seqEnd = j;
                    }
            }
    return maxSum;
}
```

**figure 5.4**
A cubic maximum contiguous subsequence sum algorithm
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */

public static int maxSubsequenceSum(int[] a)
{
    int maxSum = 0;

    for (int i = 0; i < a.length; i++)
    {
        int thisSum = 0;
        for (int j = i; j < a.length; j++)
        {
            thisSum += a[j];
            if (thisSum > maxSum)
            {
                maxSum = thisSum;
                seqStart = i;
                seqEnd = j;
            }
        }
    }

    return maxSum;
}
/**
 * Linear maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */

public static int maximumSubsequenceSum(int[] a) {
    int maxSum = 0;
    int thisSum = 0;

    for (int i = 0, j = 0; j < a.length; j++) {
        thisSum += a[j];

        if (thisSum > maxSum) {
            maxSum = thisSum;
            seqStart = i;
            seqEnd = j;
        } else if (thisSum < 0) {
            i = j + 1;
            thisSum = 0;
        }
    }

    return maxSum;
}
What does “linear runtime” really mean?

A linear function (program, algorithm) requires resources that scale linearly with the input size.

If a linear algorithm runs for 5 seconds on an input of size 10, how much time will it (approximately) run on an input of size 20?

\[ f(n) = O(n) \implies f(n) = cn \text{ for some } c \]

\[ f(2n) \approx c \cdot 2n \]

Doubling the input size roughly doubles runtime.

A quadratic algorithm runs for 5 seconds on an input of size 10, how much time will it run on an input of size 20?
Best, Worst, and Average-Case Analysis

- Best case: the minimum time for any instance of size $n$
- Worst case: the maximum time for any instance of size $n$
  - If unspecified, $O(f(n))$ means the worst case runtime
- Average case: the average time for all instances of size $n$
- Successful sequential search
  - Average case: $O(n)$
  - Worst case: $O(n)$
- Unsuccessful sequential search: $O(n)$
- Successful binary search
  - Average case: $O(\log n)$
  - Worst case: $O(\log n)$
- Unsuccessful binary search: $O(\log n)$