Introduction to the Analysis of Algorithms

Based on the notes from David Fernandez-Baca
Bryn Mawr College
CS206 Intro to Data Structures

Algorithm

• An algorithm is a strategy (well-defined computational procedure) for solving a problem, independent of the actual implementation.

ARRAY EQUALITY

Input: Two arrays A and B, of the same length and without duplicates.
Question: Do A and B contain the same elements?
Problem: Array Equality

Algorithm 1
for each position $i$ in array $A$
    if element $A[i]$ does not appear in array $B$
        return false
return true

Algorithm 2
Make a copy of both arrays and sort them
for each position $i$
    if $A[i]$ is different from $B[i]$
        return false
return true

Measurement of a Better Strategy

Which strategy is better? Some potential considerations are:

- Speed?
- Memory consumption?
- Network bandwidth?
- Easiness of implementation?
- Reusability?

The most significant for us are the first two, and we will concentrate on the first one.
Time Complexity

- The **time complexity** (or **running time**) of an algorithm is a function that describes the number of basic execution steps in terms of the *input size*.
- The time complexity abstracts the components of an algorithm's performance that depend on the algorithm itself away from those components that are machine- and implementation-dependent.

Example: Sequential Search

**SEARCH**

- **Input**: An array A of length n and a value v
- **Problem**: Determine whether A contains v.

```plaintext
i = 0;
while i < n
    if A[i] == v
        return true
    i++
return false
```

- **assignment**: 1 step
- **test**: n + 1 steps
- **test**: 1 step * n iterations of the loop
- **return**: 1 step (only once!)
- **increment**: n times
- **return**: 1 step (only once!)
Example: Sequential Search

• For the worst case, the total number of steps is $T(n) = 3n + 3$.
• The execution time for an input of length $n$ is proportional to $T(n)$.
• As $n$ gets larger, the extra “+3” becomes relatively insignificant, so the execution time is roughly proportional to $3n$.
• We can simplify this statement further and say that $T(n)$ is proportional to $n$ or linear in $n$: $f(n) = n$.
• Worst-case time complexity of this algorithm is $O(n)$, or “big-O of $n$”.

Asymptotic upper bound: $O$-notation

Definition:

$T(n)$ is $O(f(n))$ if and only if there exist positive constants $c$ and $N$ such that, for all $n \geq N$,

$$T(n) \leq c f(n)$$

$T(n)$ is $O(f(n))$ if you can multiply $f(n)$ by a (possibly large) constant ($c$) so that, asymptotically (as $n$ shoots off to infinity), $T(n)$ is completely underneath $c f(n)$. 
Claim 1. $T(n) = 3n + 3$ is $O(n)$
Proof: 
Choose $c=4$ and $N=3$. Then, for any $n \geq 3$, 
\[3n + 3 \leq 3n + n \leq 4n\]

Claim 2. $T(n) = 42n + 17$ is $O(n)$
Proof: 
Choose $c=43$ and $N=17$. Then, for any $n \geq 17$, 
\[42n + 17 \leq 43n + n \leq 44n\]

General Principle

Fact 1. Every linear function $f(n) = an + b$ is $O(n)$.
Fact 2. When using $O$ notation we can ignore constant (multiplicative) factors!

Example: $T(n) = 109n + 109$ is $O(n)$.
Set $c = 2*109$ and $N = 1$.

You can think of $O(n)$ as the class of all functions that do not grow any faster than a linear function, at least for large values of $n$. 
Array Equality, Revisited

Algorithm 1
for each position $i$ in array $A$
    if element $A[i]$ does not appear in array $B$
        return false
return true

For $i = 0$ to $n-1$, sequentially search for $A[i]$ in array $B$.

Algorithm 1 Pseudocode

```
i = 0
while i < n
    found = false
    j = 0
    while j < n
        if $a[i] = b[j]$
            found = true
            break
        ++j
    if !found
        return false
    ++i
return true
```

#Times performed

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0$</td>
<td>1</td>
</tr>
<tr>
<td>while $i &lt; n$</td>
<td>$n+1$</td>
</tr>
<tr>
<td>found = false</td>
<td>$n$</td>
</tr>
<tr>
<td>$j = 0$</td>
<td>$n$</td>
</tr>
<tr>
<td>while $j &lt; n$</td>
<td>$n \times (n+1)$ at most</td>
</tr>
<tr>
<td>if $a[i] = b[j]$</td>
<td>$n \times n$ at most</td>
</tr>
<tr>
<td>found = true</td>
<td>$n \times 1$ at most</td>
</tr>
<tr>
<td>break</td>
<td>$n \times 1$ at most</td>
</tr>
<tr>
<td>++j</td>
<td>$n \times n$ at most</td>
</tr>
<tr>
<td>if !found</td>
<td>$n$</td>
</tr>
<tr>
<td>return false</td>
<td>0</td>
</tr>
<tr>
<td>++i</td>
<td>$n$</td>
</tr>
<tr>
<td>return true</td>
<td>1</td>
</tr>
</tbody>
</table>

Total $3n^2 + 8n + 3$ at most
Upper bound of Alg. 1

Claim 1. \( T(n) = O(n^2) \)
Proof:

Choose \( c=14 (=3+8+3) \) and observe that as long as \( n \geq 1 \),
\[
3n^2 + 8n + 3 \leq 3n^2 + 8n^2 + 3n^2 = 14n^2
\]

• More generally, every quadratic function is \( O(n^2) \).
• \( O(n^2) \) is the class of all functions that asymptotically grow no faster than quadratic functions.
• Note that \( 3n+3 \) is also \( O(n^2) \). However, we are most interested in describing an algorithm using the smallest (slowest growing) big-O class that we can identify. So, it is more precise to say that \( 3n+3 \) is \( O(n) \).
• Adding the extra constant-time steps does not add to the big-O complexity.

Array Operations

• Insertion
• Searching
• Deletion
• Display

• Ordered array:
  \[
  \text{int[]} \ \text{intArray} = \{ 0, 3, 6, 9, 12, 15, 18, 21, 24, 27 \};
  \]
• Unordered array:
  \[
  \text{int[]} \ \text{intArray} = \{ 18, 0, 3, 6, 24, 9, 12, 15, 21, 27 \};
  \]
Complexity

- Linear search
  - $O(N)$
- Insertion in unordered array
  - $O(1)$
- Insertion in ordered array
  - $O(N)$
- Deletion in unordered array
  - $O(N)$
- Deletion in ordered array
  - $O(N)$

Binary Search (Ordered Arrays)

BinarySearch(A, v) // A must be sorted
n = A.size
left = 0
right = n-1

while left <= right
  mid = (left + right)/2
  if A[mid] == v
    return true
  else if v < A[mid]
    right = mid -1
  else
    left = mid +1
return false

Each iteration divides the search range [left..right] by 2.

When does the loop terminate?
- we find what we are looking for, or
- there are no more elements in the search range.

Thus, the number of iterations is bounded by the number of times we can divide $n$ by 2 before we get 1. This number is known as the log base 2 of $n$. 
Logarithms

- \(32 = 2 \times 2 \times 2 \times 2 \times 2 = 2^5\), it will take 5 iterations to get down to 1.
- The number 5 is called the log base 2 of 32. It is the exponent \(x\) such that \(2^x = 32\).
- For arbitrary \(n\), the number of iterations equals the number \(x\) of times we can divide \(n\) by half so that we get 1.
- Thus, \(x\) is the exponent for which \(n(1/2)^x = 1\). Equivalently, \(x\) is the number such that \(2^x = n\); i.e., \(x\) is the log base 2 of \(n\).
- In general \(x\) will not be a whole number but is never more than 1 away from the number of iterations.

```c
int n = 32;
while (n > 1) { n = n/2; }
```

Subset Sum

- Enumerate all subsets of the elements of \(A\). For each subset, see if its elements add up to \(K\).
- There are \(2n\) subsets to enumerate. (Why?)
- Therefore, the algorithm takes \(O(n*2^n)\) time.
- Subset Sum is \(NP-complete\), which means that it is likely not to have an efficient algorithm.
Asymptotic Analysis

Hierarchy of Function Classes

- Constant, $O(1)$, functions don’t grow at all.
- Logarithmic, $O(\log n)$, functions are slower growing than linear functions.
- Linear, $O(n)$, functions are slower growing than $O(n \log n)$ functions.
- $O(n \log n)$ functions are slower growing than quadratic functions.
- Polynomial functions, i.e., $O(n^k)$ functions where $k$ is constant.
- Exponential functions, i.e., $O(a^n)$ functions where $a > 1$. 

Big O times

[Graph showing different Big O notations]
Some General Observation

- **O(1)** denotes “constant time” – anything not dependent on the input size.
- A polynomial is always big-O of its leading term.
- For a O(f) operation followed by an O(g) operation, you can ignore the smaller one. E.g., O(n^2 + n) is O(n^2).
- If a O(f) operation is repeated O(g) times, the total time is O(f • g). E.g., if an O(n^2) operation is performed O(n \log n) times, the whole thing is O(n^3 \log n).
- If the problem size n is decreased by a **constant factor** at each step, the number of steps is O(\log n).