"Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."
- Edsger Dijkstra

What Makes a Good Algorithm?

Suppose you have two possible algorithms that do the same thing; which is better?

What do we mean by better?
- Faster?
- Less space?
- Easier to code?
- Easier to maintain?
- Required for homework?

FIRST, Aim for simplicity, ease of understanding, correctness.
SECOND, Worry about efficiency only when it is needed.

How do we measure speed of an algorithm?

Basic Step: one “constant time” operation

Constant time operation: its time doesn’t depend on the size or length of anything. Always roughly the same. Time is bounded above by some number

Basic step:
- Input/output of a number
- Access value of primitive-type variable, array element, or object field
- Assign to variable, array element, or object field
- Perform arithmetic or logical operation
- Method call (not counting arg evaluation and execution of method body)

Counting Steps

// Store sum of 1..n in sum
sum= 0;
// Inv: sum = sum of 1..(k-1)
for (int k= 1; k <= n; k= k+1){
    sum= sum + k;
}

All basic steps take time 1.
There are n loop iterations.
Therefore, takes time proportional to n.

Statement: # times done
sum= 0; 1
k= 1; 1
k <= n n+1
k= k+1; n
sum= sum + k; n
Total steps: 3n + 3

Linear algorithm in n

Not all operations are basic steps

// Store n copies of ‘c’ in s
s= "";
// Inv: s contains k-1 copies of ‘c’
for (int k= 1; k <= n; k= k+1){
    s= s + ‘c’;
}

Concatenation is not a basic step. For each k, concatenation creates and fills k array elements.

String Concatenation

\[ s = s + “c”; \] is NOT constant time.
It takes time proportional to 1 + length of s

Statement: # times done
s= s + “c”; 1
0 ‘d’ 1
1 ‘x’ 2
2 ‘e’
Not all operations are basic steps

// Store n copies of 'c' in s
s = "";
// inv: s contains k-1 copies of 'c'
for (int k = 1; k <= n; k += 1) {
    s = s + 'c';
}

Total steps: \( n^2 \) + 3

Concatenation is not a basic step. For each k, concatenation creates and fills k array elements.

Linear versus quadratic

// Store sum of 1..n in sum
sum = 0;
// inv: sum = sum of 1..(k-1)
for (int k = 1; k <= n; k += 1) {
    sum = sum + n + 1;
}

Linear algorithm

One is linear in n—takes time proportional to n
One is quadratic in n—takes time proportional to n^2

Looking at execution speed

Number of operations executed

<table>
<thead>
<tr>
<th>2n+2, n+2, n are all linear in n, proportional to n</th>
</tr>
</thead>
<tbody>
<tr>
<td>2n+2 ops</td>
</tr>
<tr>
<td>n^2 ops</td>
</tr>
<tr>
<td>Constant time</td>
</tr>
</tbody>
</table>

Linear algorithm

What do we want from a definition of “runtime complexity”?

1. Distinguish among cases for large n, not small n
2. Distinguish among important cases, like
   - n^2 basic operations
   - n basic operations
   - log n basic operations
   - 5 basic operations
3. Don’t distinguish among trivially different cases.

"Big O" Notation

Formal definition: \( f(n) = O(g(n)) \) if there exist constants \( c > 0 \) and \( N \geq 0 \) such that for all \( n \geq N \), \( f(n) \leq c \cdot g(n) \)

Prove that \( (2n^2 + n) \) is \( O(n^2) \)

Formal definition: \( f(n) = O(g(n)) \) if there exist constants \( c > 0 \) and \( N \geq 0 \) such that for all \( n \geq N \), \( f(n) \leq c \cdot g(n) \)

Example: Prove that \( (2n^2 + n) \) is \( O(n^2) \)

Methodology:

Start with \( f(n) \) and slowly transform into \( c \cdot g(n) \):
- Use \( = \) and \( \leq \) and \( < \) steps
- At appropriate point, can choose \( N \) to help calculation
- At appropriate point, can choose \( c \) to help calculation
Prove that \((2n^2 + n)\) is \(O(n^2)\)

**Formal definition:** \(f(n) = O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)

**Example:**

\[ f(n) = (2n^2 + n) = O(n^2) \]

\[ f(n) = 2n^2 + n \leq 2n^2 + n \]

Choose \(N = 1\) and \(c = 3\)

Formal definition:

\(f(n)\) is \(O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)

---

Prove that \(100n + \log n\) is \(O(n)\)

**Formal definition:** \(f(n) = O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)

\[ f(n) = 100n + \log n \leq 100n + 1 \]

Choose \(N = 1\) and \(c = 101\)

Formal definition:

\(f(n)\) is \(O(g(n))\) if there exist constants \(c > 0\) and \(N \geq 0\) such that for all \(n \geq N\), \(f(n) \leq c \cdot g(n)\)

**O(…)** Examples

Let \(f(n) = 3n^2 + 6n - 7\)

- \(f(n) = O(n^2)\)
- \(f(n) = O(n^3)\)
- \(f(n) = O(n^4)\)
- \(f(n) = O(\log n)\)
- \(f(n) = O(n)\)
- \(f(n) = O(1)\)

\(p(n) = 4n \log n + 34n - 89\)

- \(p(n) = O(n \log n)\)
- \(p(n) = O(n)\)
- \(p(n) = O(\log n)\)

\(h(n) = 20n^2 + 40n\)

- \(h(n) = O(2n^2)\)
- \(h(n) = O(n)\)

\(o(n) = 34\)

- \(o(n) = O(1)\)

**Problem-size examples**

- Suppose a computer can execute 1000 operations per second, how large a problem can we solve?

<table>
<thead>
<tr>
<th>operations</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>1000</td>
<td>60,000</td>
<td>3,600,000</td>
</tr>
<tr>
<td>(n \log n)</td>
<td>140</td>
<td>4893</td>
<td>200,000</td>
</tr>
<tr>
<td>(n^2)</td>
<td>31</td>
<td>244</td>
<td>1897</td>
</tr>
<tr>
<td>(3n^2)</td>
<td>18</td>
<td>144</td>
<td>1096</td>
</tr>
<tr>
<td>(n^3)</td>
<td>10</td>
<td>39</td>
<td>153</td>
</tr>
<tr>
<td>(2^n)</td>
<td>9</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

**Commonly Seen Time Bounds**

<table>
<thead>
<tr>
<th></th>
<th>constant</th>
<th>excellent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(1))</td>
<td>constant</td>
<td>excellent</td>
</tr>
<tr>
<td>(O(\log n))</td>
<td>logarithmic</td>
<td>excellent</td>
</tr>
<tr>
<td>(O(n))</td>
<td>linear</td>
<td>good</td>
</tr>
<tr>
<td>(O(n \log n))</td>
<td>logarithmic</td>
<td>good</td>
</tr>
<tr>
<td>(O(n^2))</td>
<td>quadratic</td>
<td>maybe OK</td>
</tr>
<tr>
<td>(O(n^3))</td>
<td>cubic</td>
<td>maybe OK</td>
</tr>
<tr>
<td>(O(2^n))</td>
<td>exponential</td>
<td>too slow</td>
</tr>
</tbody>
</table>

**Do NOT say or write** \(f(n) = O(g(n))\)

**Mathematically, it is a disaster.**

You see it sometimes, even in textbooks. Don’t read such things.

Here’s an example to show what happens when we use = this way.

We know that \(n+2\) is \(O(n)\) and \(n+3\) is \(O(n^2)\). Suppose we use:

\[ n+2 = O(n) \]

\[ n+3 = O(n^2) \]

But then, by transitivity of equality, we have \(n+2 = n+3\). We have proved something that is false. Not good.
Java Lists

- `java.util` defines an interface `List<E>` implemented by multiple classes:
  - `ArrayList`
  - `LinkedList`

Linear search for `v` in `b[0..]`

// Store value in `i` to truthify `b[0..i-1] < v <= b[i..]
// Precondition: `b` is sorted

| pre: `b` | sorted |
| post: `b` | `< v` | `≥ v` |
| inv: `b` | `< v` | sorted |

Methodology:
1. Define pre and post conditions.
2. Draw the invariant as a combination of pre and post.
3. Develop loop using 4 loopy questions.
Practise doing this!

Linear algorithm: O(b.length)

Linear algorithm: O(b.length)

Binary search for `v` in `b[0..]`

// Store value in `i` to truthify `b[0..i-1] < v <= b[i..]
// Precondition: `b` is sorted

| pre: `b` | sorted |
| post: `b` | `< v` | `≥ v` |
| inv: `b` | `< v` | sorted |

Methodology:
1. Define pre and post conditions.
2. Draw the invariant as a combination of pre and post.
3. Develop loop using 4 loopy questions.
Practise doing this!
Binary search for v in b[0..]

// Store value i to truthify b[0..i-1] < v <= b[i..]  
// Precondition: b is sorted  

0  b.length  k= -1;  
i= b.length;  

pre: b sorted  

0  k  i  b.length  
inv: < v | sorted | ≥ v  

Make invariant true initially

Dutch National Flag Algorithm

This algorithm is better than binary searches that stop when v is found.  
1. Gives good info when v not in b.  
2. Works when b is empty.  
3. Finds first occurrence of v, not arbitrary one.  
4. Correctness, including making progress, easily seen using invariant  

Logarithmic: O(log(b.length))

while (k < i - 1) {  
  int j=(k+i)/2;  
  // k < j < i  
  Set one of k, i to j  
  
  if (b[j] < v)  
    k= j;  
  else  
    i= j;  
}

Determine loop condition B:  
!B && inv imply post

Figure out how to make progress toward termination.  
Cut size of b[k+1..i-1] in half
### Dutch National Flag Algorithm

Swapping the colors of the Dutch national flag can be modeled as a sorting problem. The algorithm involves swapping the values of the array `b[0..n-1]` to put the reds first, then the whites, then the blues. This is done by first truthifying the postcondition given the initial preconditions.

#### Invariant P1

- **Q:** `b[0..n]`
- **R:** `b[0..n]`
- **P1:** `b[0..n]`

#### Invariant P2

- **Q:** `b[0..n]`
- **R:** `b[0..n]`
- **P2:** `b[0..n]`

### Asymptotically, which algorithm is faster?

- **Invariant 1**
- **Invariant 2**

---

**Dutch National Flag Algorithm:** invariant P1

- **Q:** `b[0..n]`
- **R:** `b[0..n]`
- **P1:** `b[0..n]`

### Asymptotically, which algorithm is faster?

- **Invariant 1**
- **Invariant 2**

---

**Dutch National Flag Algorithm:** invariant P2

- **Q:** `b[0..n]`
- **R:** `b[0..n]`
- **P2:** `b[0..n]`

---

**Asymptotically, which algorithm is faster?**

- **Invariant 1**
- **Invariant 2**

---

**Dutch National Flag Algorithm**

Swapping `b[0..n-1]` to put the reds first, then the whites, then the blues. That is, given precondition `Q`, swap values of `b[0..n]` to truthify postcondition `R`:

- **Q:** `b[0..n]`
- **R:** `b[0..n]`
- **P1:** `b[0..n]`
- **P2:** `b[0..n]`

---

**Dutch National Flag Algorithm: invariant P1**

- **Q:** `b[0..n]`
- **R:** `b[0..n]`
- **P1:** `b[0..n]`

---

**Dutch National Flag Algorithm: invariant P2**

- **Q:** `b[0..n]`
- **R:** `b[0..n]`
- **P2:** `b[0..n]`

---

**Asymptotically, which algorithm is faster?**

- **Invariant 1**
- **Invariant 2**

---