TIME ANALYSIS OF ALGORITHMS WITHOUT RECURSION

Big differences

We have seen that, for a given problem, one algorithm may be vastly more efficient than others. Examples:

Searching a list of \( n \) elements:
- linear search takes time on the order of \( n \).
- binary search takes time on the order of \( \log_2 n \).

Finding the cheapest paving in a graph with \( n \) nodes:
- brute force takes time on the order of \( 2^{n^2} \).
- greedy algo takes time on the order of \( n^2 \).

[ Is “on the order of” bothering you? Good — we’ll define things properly very soon.]

Small differences

We can sometimes fine tune a given algorithm to make it a little faster.

Example: Linear search with a dummy record is faster than ordinary linear search because there is less work per iteration.

(But any kind of linear search takes takes time on the order of \( n \) because there are roughly \( n \) iterations in the worst case.)

Time efficiency

We like to know the time efficiency of a program for several reasons:

- To get an estimate of how long a program will run. (Waiting for it to finish may not be feasible!)

- To get an estimate of how large an input the program can handle without taking too long.

- To compare the efficiency of different programs for solving the same problem.

We could run timing tests on the program, but we prefer to analyze the efficiency of the algorithm — before we’ve written the program.

Why?
Big differences are the first priority

When analyzing algorithms for a given problem, it makes sense to pay attention to the big differences first.

- Example: We would choose the greedy strategy for graph paving over the brute force strategy.

Only then does it make sense worry about the small differences.

- Example: There is no sense in fine tuning the brute force algorithm. Any effort towards fine tuning should be spent on the greedy algorithm.

What we need

We need a technique for analyzing algorithm efficiency that:

- is precise about what we mean by “on the order of”
- can distinguish the big differences
- (ideally,) allows for quick and easy analysis

Solution: We will learn a technique for estimating time efficiency to within a constant factor.

Because we ignore the constant factors, the analysis is easy. But it still makes the big distinctions.

Ignoring constant factors

Consider two programs $A$ and $B$ for solving a given problem, with running times of

$T_A(n) = n^3$ and $T_B(n) = 8n + 3$.

Which program is faster?

For inputs of size less than 3: program $A$. For inputs of size greater than 3: program $B$. $n = 3$ is called the breakpoint.

B is eventually superior no matter what the constants

What if program $A$ were a million times faster and program $B$ a million times slower, i.e., if:

$T_A(n) = n^3/1,000,000$ and $T_B(n) = (8n + 3) * 1,000,000$.

It would still be true that:

- $B$ would eventually be faster than $A$, and
- $B$'s superiority would grow as $n$ increases.

(The breakpoint would change, however.)

This is true no matter what the constants are!

Conclusion: For large values of $n$, the form of a mathematical function has more effect on its growth rate than a constant multiple.
Growth rates of various functions

<table>
<thead>
<tr>
<th>function</th>
<th>$T(n)$ for $n =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log n$</td>
<td>10 100 1,000 10,000 10^5</td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
<td>3 10 31 100 316</td>
</tr>
<tr>
<td>$n$ log $n$</td>
<td>10 100 1,000 10,000 10^5</td>
</tr>
<tr>
<td>$n^2$</td>
<td>100 10,000 10^6 10^6 10^6</td>
</tr>
<tr>
<td>$2^n$</td>
<td>10^24 10^30 10^300 10^3000 10^30000</td>
</tr>
</tbody>
</table>

There is a computational cliff when we reach “exponential” functions: one in which the variable appears in the exponent.

To get a sense of scale:
- there are $10^{43}$ atoms in the universe
- there have been $10^{17}$ seconds since the big bang

If we can perform 1 billion operations per second,
- $10^{16}$ operations take 1 year
- $10^{20}$ operations take 10,000 years!

Would it help if we could do 100 million billion per second? How about $10^{30}$?

77

Towards a Definition

Say we have an algorithm (or program) whose running time on an input of size $n$ is $f(n)$.

We don’t know what $f(n)$ is, but we want to estimate it to within a constant factor.
Let’s call that estimate $g(n)$.

We would be happy with our estimate $g(n)$ even if the relationship between $g(n)$ and $f(n)$ holds only after some breakpoint $B$.

In other words, from $B$ onwards, we want $g(n)$ to estimate an upper bound on $f(n)$, to within a constant factor.

That is, we want there to be some constant factors $c$ and $B$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq B$.

We don’t care what these constants are; we just need them to exist.

79

O Notation, or “big-oh” notation

Consider any 2 functions $f, g$ defined on the nonnegative integers $N = \{0, 1, 2, \ldots\}$ such that $f(n), g(n) \geq 0$ for all $n \in N$.

Definition: $f(n)$ is $O(g(n))$ if there exist positive constants $c$ and $B$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq B$.

This means that, to within a constant factor, $f(n)$ grows no faster than $g(n)$.

We pronounce this:

- $f$ has order $g$, or
- $f$ is “oh” or “big-oh” of $g$
2 Key Properties of $O$ Notation

**Constant factors disappear**

If $d > 0$ is a constant, then $df(n)$ is $O(f(n))$ and $f(n)$ is $O(df(n))$.

Examples:
- $6n$ and $\frac{n}{2}$ are $O(n)$.
- $n$ is $O(29n)$ and $O(642n)$.

**Low-order terms disappear**

If $\lim_{n \to \infty} \frac{h(n)}{g(n)} = 0$ then $g(n) + h(n)$ is $O(g(n))$.

Examples:
- $n^5 + n^3 + 6n^2$ is $O(n^5)$.
- $n^2 + n(\log n)^3$ is $O(n^2)$.

---

Proving a Big-oh Bound

**Example**

Prove: $6n + 3$ is $O(n)$.

Proof:
- Let $c = 7$ and $B = 3$
  - $6n + 3 \leq 7 \cdot n$ for all $n \geq 3$, so
  - $6n + 3 \leq c \cdot n$ for all $n \geq B$.
- So there exist $c$ and $B$ such that $6n + 3 \leq c \cdot n$ for all $n \geq B$.
- So $6n$ is $O(n)$, by definition of big-oh.

**Exercise:** Prove that the following function is $O(n^3 \cdot n \log n)$:

$$f(n) = (6n^3 + n \log n + 56) \cdot (73n \log n + 10^5)$$

---

$O$ Notation Defines Sets

$6n$ is $O(n)$ and $O(3n)$ and $O(2^n)$.

In fact, it is all of these:

- $O(2^n)$
- $O(n^3)$
- $O(6n - 99)$
- $O(3n)$
- $O(n + 8)$
- $O(n)$

But it is not any of these:

- $O(\log n)$
- $O(\sqrt{n})$

How can this be?

If some constant times $n$ is an upper bound on our function (after some point $B$), then certainly some constant times $3n$ will be.

$O(n)$, e.g., defines a set containing all mathematical functions that are of that order.

Note that we always look for the smallest ("tightest") and simplest upper bound function that will satisfy the big-oh criteria.

E.g., for $6n$, $O(n)$ is the smallest upper bound instead of, say, $O(n^2)$, and is also the simplest description of it instead of, say, $O(6n + 22)$. 
Remember this

\[ O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n) \subseteq O(3^n) \subseteq O(n!) \subseteq O(n^n) \]

We can use this to determine which term in a mathematical function is the most dominant, and which other terms can be "cancelled".

Examples
- \[ O(5 \log n + n^2 + \frac{2n^3}{3}) \]
- \[ O(12n + n \log n) \]
- \[ O(12n + n \log n + 2^n + n^2) \]

Using \( O \) notation to Analyze the Running Time of Programs

Using very simple techniques, we can analyze code and know that the time to execute it is, for example,

\[ O(n^2) \]

without having to know that the more detailed answer is, for example,

\[ \frac{n^2 + 7 \log n}{3} \]

Example

```java
static void silly (float num) {
    num = 0;
    num = (float) Math.sqrt(667.2);
    num = num / .000931f;
    System.out.println("num is " + num);
    System.out.println("Bye!");
}
...
if (n > 1000)
    System.out.println("That's big");
else
    System.out.println("That's not so big");
silly(n);
```

Example

```java
for (int i=100; i<=n; i++)
    sum++;
```

Analysis:
- This code involves no loops or recursion.
- Therefore it takes a constant amount of time, for some unknown constant.
- We call this "constant time" or \( O(1) \).
Example

for (int i=1; i<=n/2; i++)
    for (int j=1; j<=n*n; j++)
        sum++;

Analysis (working from the inside out):

- Each iteration of the inner loop takes $O(1)$ time.
- On every iteration of the outer loop, $O(n^2)$ iterations of the inner loop are performed.
- Thus each iteration of the outer loop takes $O(n^2)$ time.
- The outer loop is performed $\lceil n/2 \rceil$ times and $\lceil n/2 \rceil$ is $O(n)$.
- Therefore the loop takes $O(n^3)$ time.
- Therefore the program takes $O(n^3 + 1)$ time. (1 is for the initialization.)
- Thus the entire program takes $O(n^3)$ time.

Write your analyses in this style. Annotating the code is not sufficient.

Example

sum = 0;
for (int i=1; i<=n/2; i++)
    sum++;

for (int j=1; j<=n*n; j++)
    sum++;

Example

if (n % 2 == 0)
    for (int j=1; j<=n*n; j++)
        sum++;
else
    for (int k=5; k<=(n+1); k++)
        sum += k;

Example

static boolean isSorted (int[] List) {
    // Details omitted
}

int[] myList = new int[size];
sum = 0;
if (isSortedList)
    for (int j=1; j<=n*n; j++)
        sum++;
else
    for (int k=5; k<=(n+1); k++)
        sum += k;
Example

static void blah (int n) {
    int sum = 0;
    for (int i=1; i<=(n/2); i++)
        for (int j=1; j<=n; j++)
            sum++;
    System.out.println (sum);
}

blah (p)     blah (j*j)

Example

sum = 0;
while (num > 1) {
    num = num / 2;
    sum++;
}

Example

for (int k=1; k<=5000; k++)
    if ( A[k] % 2 == 0 )
        even++;
    else
        odd++;
Example

```java
sum = 0;
for (int i=1; i<=n; i++)
    for (int j=1; j<=i; j++)
        sum++;
}
```

Analysis:

- Each iteration of the inner loop takes $O(1)$ time.
- On the $i$th iteration of the outer loop, $i \leq n$ iterations of the inner loop are performed.
- Thus each iteration of the outer loop takes $O(n)$ time.
- The outer loop is performed $n$ times.
- Thus the entire program takes $O(n^2)$ time.

Overestimating

$O(n^2)$ seems to be an overestimate.

The first time we do the inner loop, it iterates only once. On subsequent visits to the inner loop, it iterates more and more times, until finally it does $n$ iterations.

The total number of iterations of the inner loop is $1 + 2 + \cdots + n = n(n + 1)/2$.

However, $n(n + 1)/2$ is $O(n^2)$, so we get the same final answer, in terms of big-oh.

In some cases, the extra precision in the answer $n(n + 1)/2$ may be important.

---

Summary of Rules

<table>
<thead>
<tr>
<th>Type of code</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>no loops or recursion:</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>loop:</td>
<td>multiply</td>
</tr>
<tr>
<td>sequence:</td>
<td>add</td>
</tr>
<tr>
<td>selection:</td>
<td>take the maximum</td>
</tr>
<tr>
<td>method call:</td>
<td>apply method’s time to size of arguments</td>
</tr>
</tbody>
</table>