CSCI 2011: Algorithms and Big-O Analysis

Chris Kauffman

Last Updated:
Wed Jul 4 14:01:45 CDT 2018

Logistics

Reading: Rosen

- Now: 3.1 - 3.3
- Next: 4.1 - 4.6

Assignments

- A04: post later today
- Due Tuesday

Quizzes

- Quiz 02 today

Goals

- Finish discrete structures
- Discuss Algorithms and Big-O analysis

Algorithms

- Finite sequence of concrete steps to take to accomplish something
  - Finite in that you can write the steps down in a finite amount of space, **not** that the algorithm will terminate in a finite amount of time
- After two semesters of programming should have been exposed to a variety of algorithms including
  - **Searching** for elements in data structures like lists, arrays, trees. Might be a `find_max(a[])` or a `find_query(q,x[])`
  - **Sorting** elements in arrays like `bubble_sort(a[])` or `insertion_sort(a[])`, possibly also more efficient versions like `merge_sort(a[])`
- Will use real code and **pseudocode**, code-like but informal

```
find_max(a[] : integer array):  // find max in array a[]
  max = a[1]
  for i=2 to length(a):
    if max < a[i]:
      max = a[i]
end
```

Exercise: Counting Operations

- The **execution time** of an algorithm depends on the number of **operations** performed
- Operations may take a varying times but can rely on common operations taking a fixed duration
  - Adding two small-ish numbers or incrementing
  - Comparing two numbers
  - Accessing an array element
- To that end, **counting** the number of operations performed approximates execution time

```
find_max(a[] : integer array):
  max = a[1]
  for i=2 to length(a):
    if max < a[i]:
      max = a[i]
end
```

Count the **maximum** operations performed for `find_max()` for input arrays sized

- 4
- 10
- N

Exercise: Counting Operations

```
find_max(a[] : integer array):
  max = a[1]                  # 1, setting variable
  for i=2 to length(a):       # 2, set i / increment, compare to length
    if max < a[i]:            # 2, access and compare
      max = a[i]              # 2, maybe access and update
end
```

Count the **maximum** operations performed for `find_max()` for input arrays sized

- `4 : 1 +  4*(2+2+2) = 25`
- `10: 1 + 10*(2+2+2) = 61`
- `N:  1 +  N*(2+2+2) = 6N+1`
- These are maximum counts as NOT hitting conditional may lower it
- Omits some details of control like operations required to track the next instruction

Approximating Algorithm Complexity

- Early on folks measured runtimes against one another
- Problematic as if my CPU can do addition in half the time as yours, my results might look better even though my algorithm is worse
- To properly compare algorithms, count steps
- Still problematic as details what operations vary a bit between CPUs
- Approximate this with **Order Notation** which describes roughly how performance scales with input size

```
find_max(a[] : int array):
  max = a[1]
  for i=2 to length(a):
    if max < a[i]:
      max = a[i]
end
```

- `find_max(a[])` will take a maximum of $6N+1$ steps to complete
- Its execution time scales **linearly** with its input size
- `find_max(a[])` has runtime $O(N)$

It's Show Time!

Not *The* Big O

Just Big O

Let $f(x)$ be a function of x (integer or real).
$f(x)$ is $O(g(x))$ if there are positive constants C and k such that

- When $x > k$
- $|f(x)| \leq C|g(x)|$

Reads

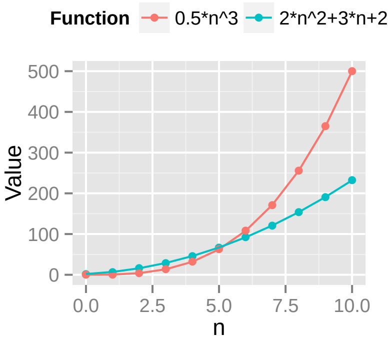
- "$f(x)$ is big-O $g(x)$"
- g grows at least as fast as f
- "f is upper bounded by g"

Show It

Show that
$f(n) = 2n^2 + 3n + 2$ is $O(n^3)$

- Pick $C = 0.5$ and $k = 6$
- Called **witnesses**

n	$f(n)$	$0.5n^3$	
0	2	0	
1	7	0	
2	16	4	
3	29	13	
4	46	32	
5	67	62	
6	92	108	$\leftarrow$ **k**
7	121	171	



How about the opposite? Show

$$g(n) = n^3 \text{ is } O(2n^2 + 3n + 2)$$

Exercise: Can I get a Witness?

Show that $7x^2$ is $O(x^3)$.
Do so by finding witnesses C, k such that $7x^2 \leq Cx^3$ for all $x > k$.

Answers: Can I get a Witness?

Show that $7x^2$ is $O(x^3)$. Do so by finding witnesses C, k such that $7x^2 \leq Cx^3$ for all $x > k$. Show that $7x^2$ is $O(x^3)$.
Do so by finding witnesses C, k such that $7x^2 \leq Cx^3$ for all $x > k$.

- Divide both sides of inequality by x^2
- Equivalent inequality $7 < x$
- When $x > 7$, we have $7x^2 < x^3$
- Witnesses exist: $C = 1, k = 7$ ■

Showing Something Isn't $O(g(x))$

Show that n^2 is not O(n): no pair of witnesses C and k exist such that $n^2 \leq Cn$ whenever $n > k$.

Proof by Contradiction:

1. Suppose C, k do exist.
2. Divide both sides of the inequality $n^2 \leq Cn$ by n to get $n \leq C$.
3. No matter what C and k are, inequality $n \leq C$ doesn't hold for $n = max(k+1, C+1)$
4. Contradiction shows means that that n^2 is not $O(n)$. ■

Most proofs of "not big-O" follow a similar flavor.

Combining functions in big-O

- $a(n)$ is $O(x(n))$
- $b(n)$ is $O(y(n))$

Combine functions a, b by

ADDING	$a(n) + b(n)$	is $O(x(n) + y(n))$
MULTIPLYING	$a(n) \cdot b(n)$	is $O(x(n) \cdot y(n))$

Special Cases

$a(n)$ and $b(n)$	BOTH	$O(x(n))$	then	$a(n) + b(n)$	is	$O(x(n))$
$a(n)$ and $b(n)$	BOTH	$O(x(n))$	then	$a(n) \cdot b(n)$	is	$O(x(n)^2)$

Constant Time Program Operations

The following take O(1) Time (Constant Time)

- Arithmetic operations (add, subtract, divide, modulo)
  - Integer ops usually practically faster than floating point
- Accessing a stack variable
- Accessing a field of an object
- Accessing a single element of an array
- Doing a **primitive** comparison (equals, less than, greater than)
- *Calling* a function/method but **NOT waiting for it to finish**

The following take more than O(1) time (how much more)?

- Raising an arbitrary number to arbitrary power
- Allocating an array
- Checking if two `Strings` are equal
- Determining if an array or `ArrayList` `contains()` an object

Common Code Patterns

- Adjacent Loops Additive: $2 \times n$ is $O(n)$

```
for(int i=0; i<N; i++){
  blah blah blah;
}
for(int j=0; j<N; j++){
  yakkety yack;
}
```

- Nested Loops Multiplicative usually polynomial
  - 1 loop, $O(n)$
  - 2 loops, $O(n^2)$
  - 3 loops, $O(n^3)$
- Repeated halving usually involves a logarithm
  - Binary search is $O(\log n)$
  - Fastest sorting algorithms are $O(n \log n)$
  - Proofs are harder, require solving recurrence relations

Lots of special cases so be careful

Exercise: Complexity of Reversal

Two functions to reverse an array. Discuss

- Big-O estimates of **runtime** of both
- Big-O estimates of **memory overhead** of both
  - Memory overhead is the amount of memory in addition to the input required to complete the method
- Which is practically better?

reverseE

```
void reverseE(Integer a[]){
  int n = a.length;
  Integer b[] = new Integer[n];
  for(int i=0; i<n; i++){
    b[i] = a[n-1-i];
  }
  for(int i=0; i<n; i++){
    a[i] = b[i];
  }
}
```

reverseI

```
reverseI(Integer a[]){
  int n = a.length;
  for(int i=0; i<n/2; i++){
    int tmp = a[i];
    a[i] = a[n-1-i];
    a[n-1-i] = tmp;
  }
  return;
}
```

Exercise: Much Trickier Allocation Exercise

```
// Concatenate all strings in arr
//   concat_all({"A","B","C","D","E"})
// results in "ABCDE"
string concat_all(string arr[]) {
  string result = "";
  for(int i=0; i<length(arr); i++){
    result = result + arr[i];
  }
  return result;
}
```

- Give a Big-O estimate for the runtime
- Give a Big-O estimate for the memory overhead

Answers: Much Trickier Allocation Exercise

- Cannot alter size of allocated memory blocks
- `result = result + arr[i];` Creates a new string which combines two existing strings
  - Allocate combined space
  - Copy all characters from both, implicit loops
  - Redirect pointer for result
- Runtime Complexity: $O(n^2)$ for strings length 1
- Space Complexity: $O(n)$ at least, worse if not garbage collecting effectively

```
// Concatenate all strings in arr
//   concat_all({"A","B","C","D","E"})
// results in "ABCDE"
string concat_all_verbose(string arr[]){
  string result = "";
  for(int i=0; i<length(arr); i++){
    int size =
      length(result) + length(arr[i]);
    string tmp = new string[size];
    for(int j=0; j<length(result); j++){
      tmp[j] = result[j];
    }
    for(int j=0; j<length(arr[i]); j++){
      tmp[j+length(result)] = arr[i][j];
    }
    free(result);
    result = tmp;
  }
  return result;
}
```

Proper data structures / algorithm gets linear runtime

Exercise: Hash Code Efficiency

- **Hash code**: an integer computed from some object such as a character string; used to place it in a hash table
- NOT an invertible computation: cannot map from number to string
- Spreads well: changing single characters changes hash code considerably
- Typically computed using the following functions *similar* to the right

```
// Computes hash code =
// s[0]*31^(n-1) + s[1]*31^(n-2) +
// ...
// + s[n-3]*31^2 s[n-2]*31 + s[n-1]

int hash_code(char array str[]){
  int hc = 0;
  int n = length(str);
  for(int i=0; i<n; i++){
    hc = hc + str[i] * pow(31,n-i-1));
  }
  return h;
}
```

- What is the Big-O Time Complexity of this function?
- State any assumptions about functions that are used
- Could you improve the efficiency of this code?

Answers: Hash Code Efficiency

- Complexity hinges the `pow(x,y)` function
- `pow(x,y)` is NOT $O(1)$: try computing 3^{25} a see how long it takes you
- If `pow(x,y)` is $O(n)$, `hash_code(str)` is $O(n^2)$
- We will see a version of `pow(x,y)` which is $O(\log_2 n)$ making `hash_code(str)` $O(n \log_2 n)$
- `hash_code_linear(str)` is $O(n)$: no hidden nested loops

```
// LINEAR TIME hash code computation
int hash_code_linear(char str[]){
  int hc = 0;
  int base = 31;
  int power = 1;
  int n = length(str);
  for(int i=n-1; i>=0; i--){
    h = h + str[i] * power;
    power = power * base;
  }
  return h;
}
```

Enrichment: Find a way to get $O(n)$ iterating from low to high index in `str`

Exercise: Binary Search Complexity

- Typical code for iterative binary search
- What is its complexity in big-O terms?
- How could one **prove** this?

```
int binary_search(int a[], int key){
  int left=0, right=a.length-1;
  int mid = 0;
  while(left <= right){
    mid = (left+right)/2;
    if(key == a[mid]){
      return mid;
    }else if(key < a[mid]){
      right = mid-1;
    }
    else{
      left = mid+1;
    }
  }
  return -1;
}
```

Answers: Binary Search Complexity

Binary search has worst case runtime complexity $O(\log_2 N)$

Semi-formal Proof

1. Without loss of generality, assume, assume $N = 2^a$.
2. The worst case performance is when an element is not present
3. Each step of the algorithm examines the current "middle" element, eliminates it and half of remaining array
4. Performing a steps will reduce array to empty so algorithm terminates in $O(a)$ steps
5. Taking the log of $N = 2^a$ gives $a = \log_2 N$
6. By (4) and (5), binary search is $O(\log_2 N)$

Notes

- Looseness around halving vs. (halving-1)
- Will prove this again once the **Master Theorem** for recurrence relations is in hand

Bounding Functions

- Big-O: **Upper** bounded by ...
  - $2n^2 + 3n + 2$ is $O(n^3)$ and $O(2^n)$ and $O(n^2)$
- Big-Omega: **Lower** bounded by ...
  - $2n^2 + 3n + 2$ is $\Omega(n)$ and $\Omega(\log(n))$ and $\Omega(n^2)$
- Big-Theta: **Upper and Lower** bounded by
  - $2n^2 + 3n + 2$ is $\Theta(n^2)$ - **tightly bounded**
- Little-O: **Upper** bounded by **but not lower** bounded by...
  - $2n^2 + 3n + 2$ is $o(n^3)$

Big-O versus Big-Theta Jargon

- Often folks say "That algorithm is Big-O N-squared"
  - upper bounded by N^2
- Most often they **mean** "That algorithm is Big-Theta N-squared"
  - upper and lower/tightly bounded by N^2
- Kauffman will almost *always* do this

Growth Ordering of Some Functions

Name	Lead Term	Big-Oh	Example
Constant	$1, 5, c$	$O(1)$	$2.5, 85, 2c$
Log-Log	$\log(\log(n))$	$O(\log\log n)$	$10 + (\log\log n + 5)$
Log	$\log(n)$	$O(\log(n))$	$5\log n + 2$
			$\log(n^2)$
Linear	n	$O(n)$	$2.4n + 10$
			$10n + \log(n)$
N-log-N	$n\log n$	$O(n\log n)$	$3.5n\log n + 10n + 8$
Super-linear	$n^{1.x}$	$O(n^{1.x})$	$2n^{1.2} + 3n\log n - n + 2$
Quadratic	n^2	$O(n^2)$	$0.5n^2 + 7n + 4$
			$n^2 + n\log n$
Cubic	n^3	$O(n^3)$	$0.1n^3 + 8n^{1.5} + \log(n)$
Exponential	a^n	$O(2^n)$	$8(2^n) - n + 2$
		$O(10^n)$	$100n^{500} + 2 + 10^n$
Factorial	$n!$	$O(n!)$	$0.25n! + 10n^{100} + 2n^2$

Bottom category referred to as **intractable**

Problem Classes

Problem Size Described by N

Tractable Best known algorithms that solve the problem have runtime complexity $O(a^N)$ for constant a and small value of N like $N < 100$

Intractable Not Tractable: best known algorithms have runtime complexity that is a large polynomial, is exponential ($O(2^N)$), or is factorial $O(N!)$

Unsolvable No algorithm exists to solve the problem.

Most algorithms you study in 4041 will be in Tractable but there are quite a few Intractable problems that arise in the real world, particularly for salespeople.

P vs NP vs NP-Complete

Problem Class P We know a polynomial time algorithm that runs on a deterministic computer to solve it. Ex: Matrix Mult.

Problem Class NP We know a polynomial time algorithm that runs on a **non-deterministic** computer to solve it. Can **verify a solution** in polynomial time. Ex:

Non-deterministic Computer Roughly, employs an infinite number of CPUs to try many possible solutions at once. Unfortunately reality dictates that we can't easily build a non-deterministic machine and simulating one takes exponential time.

NP Problems are Difficult Best algs have worst-case performance that is exponential on problem size.

Problem Class NP-Complete Group of NP problems that can used to solve other NP problems. A polynomial time alg for any one NPC would solve all NP problems in poly-time. Ex: Boolean Satisfiability (3SAT).

A Problematic Problem

Write a program H that

1. Takes as input the code for another program P and input for P called I
2. Determines if P will eventually terminate on input I
3. Prints out "Halts" if it will terminate.
4. Prints out "Runs Forever" otherwise.

This specification is referred to as the **Halting Problem**
Propose some solutions (?!?)

No Algorithm Solves The Halting Problem

- Naive approach: Run program P on input I.
- If P finishes, print "Halts"
- If it doesn't finish... wait... crap...

Proof by Contradiction, Courtesy of Alan Turing, 1936

1. Suppose H exists which solves the halting problem
2. Notate H(P,I) to mean run H on program P with input I
3. Note that programs can be input so H(P,P) is valid
4. Let K(P) be the algorithm
   - a) Run H(P,P)
   - b) If the output is "Runs Forever", output "Halts"
   - c) If the output is "Halts", **enter an infinite loop**
5. Note that H(K,K) $\equiv$ K(K) and should give same output.
6. Suppose H(K,K) outputs "Runs Forever": K terminates on input K.
7. By code of K(K), did step (c) so in step (a), H(K,K) must output "Halts"
8. Output of H(K,K) is contradictory in 6 and 7. ∎