Asymptotic running time of algorithms

Asymptotic Complexity: leading term analysis

Comparing searching and sorting algorithms: technique so far

1. Count worst-case number of comparisons as a function of input size.
2. Drop lower-order terms and floors/ceilings to come up with asymptotic running time of algorithm.

We will now see how to generalize to other programs.

1. Count worst-case number of operations executed by program as a function of input size.
2. Use formal definition of big-O complexity to derive asymptotic running time of algorithm.

Formal definition of O() notation:

Let $f(n)$ and $g(n)$ be functions. We say that $f(n)$ is *of order* $g(n)$, written $O(g(n))$ if there is a constant $c > 0$ such that for all but a finite number of positive values of n,

$f(n) \leq c * g(n)$

In other words, $g(n)$ sooner or later overtakes $f(n)$ as n gets large.

Example: $f(n) = n + 5, g(n) = n$. We show that $f(n) = O(g(n))$.

Choose $c = 6$:

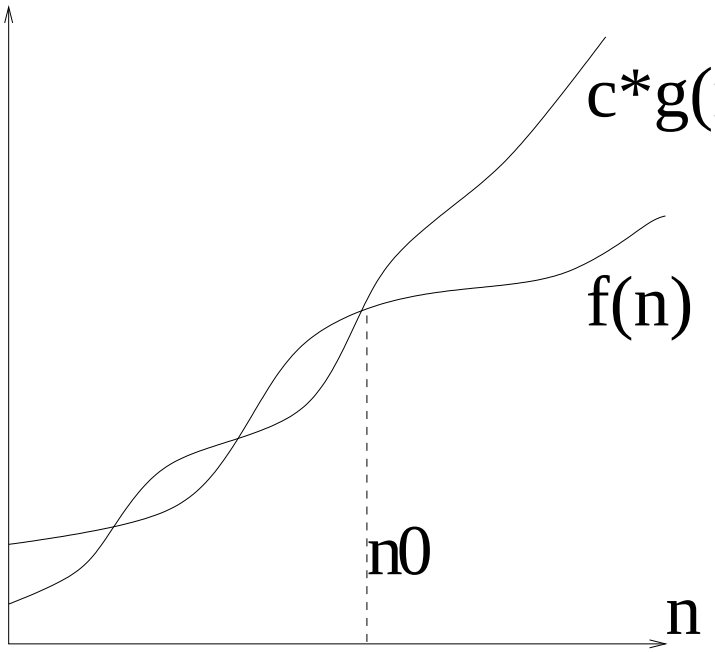
$f(n) = n + 5 \leq 6 * n$ for all $n > 0$.

Example: $f(n) = 17n, g(n) = 3n^2$. We show that $f(n) = O(g(n))$.

Choose $c = 6$:

$f(n) = 17n \leq 6 * 3n^2$ for all $n > 0$.

A graphical view of big-O notation



f(n) = O(g(n))

To prove that $f(n) = O(g(n))$, find an $n0$ and c such that $f(n) \leq c * g(n)$ for all $n > n0$. We will call the pair ($n0$,c) a *witness pair* for proving that $f(n) = O(g(n))$.

For asymptotic complexity, the base of logarithms does not matter.

Let us show that $log_2(n) = O(log_b(n))$ for any $b > 1$.

So we need to find a (c, n_0) such that

$log_2(n) \leq c * log_b(n)$ for all $n > n_0$.

Choose $(c = log_2(b), n_0 = 0)$.

This works because $c * log_b(n) = log_2(b) * log_b(n) = log_2(n)$ for all positive n.

Asymptotic complexity gives an idea of how rapidly space/time requirements grow as problem size grows.

Suppose we have a computing device that can execute 1000 operations per second. Here is the size of the problem that can be solved in a second, a minute and an hour by algorithms of different asymptotic complexity.

Complexity	1 second	1 minute	1hour
n	1000	60,000	3,600,000
n log n	140	4893	200,000
n^2	31	244	1897
$3n^2$	18	144	1096
n^3	10	39	153
2^n	9	15	21

For searching and sorting algorithms, you can usually determine big-O complexity by counting comparisons.

Reason: you usually end up doing some fixed number of arithmetic/logical operations per comparison.

Detailed counting: estimate number of SaM-like operations

- Basic operation: arithmetic/logical operation counts as 1 operation
- Assignment: counts as 1 operation (operation count of righthand side expression is determined separately)
- Loop: number of operations/iteration * number of loop iterations
- Method invocation: number of operations executed in invoked method

Example: selection sort

```
public static void selectionSort(Comparable[] a) { //array of size n
 for (int i = 0; i < a.length; i++) {      <-- cost = c1, n times
   int MinPos = i;                         <-- cost = c2, n times
   for (int j = i+1; j < a.length; j++) { <-- cost = c3, n*(n-1)/2 times
    if (a[j].compareTo(a[MinPos]) < 0)     <-- cost = c4, n*(n-1)/2 times
        MinPos = j;}                       <-- cost = c5, n*(n-1)/2 times
   Comparable temp = a[i];                 <-- cost = c6, n times
   a[i] = a[MinPos];                       <-- cost = c7, n times
   a[MinPos] = temp;}}                     <-- cost = c8, n times
```

Total number of operations = (c1+c2+c6+c7+c8)*n +
(c3+c4+c5)*n*(n-1)/2

= (c1+c2+c6+c7+c8 -(c3+c4+c5)/2)*n + (c3+c4+c5)/2 *n*n

= $O(n^2)$

Matrix multiplication

```
int n = A.length;                          <-- cost = c0, 1 time
for (int i = 0; i < n; i++) {              <-- cost = c1, n times
   for (int j = 0; j < n; j++) {           <-- cost = c2, n*n times
      sum = C[i][j];                       <-- cost = c3, n*n times
      for k = 0; k < n; k++)               <-- cost = c4, n*n*n times
         sum = sum+A[i][k]*B[k][j];        <-- cost = c5, n*n*n times
      C[i][j] = sum;                       <-- cost = c6, n*n times
   }
}
```

Total number of operations = c0 + c1*n + (c2+c3+c6)*n*n + (c4+c5)*n*n*n

$= O(n^3)$

Remarks

- For asymptotic running time, we do not need to count precise number of operations executed by each statement, provided that number is independent of input size. Use symbolic constants like c1,c2, etc.
- Our estimate used a precise count for the number of times the j loop was executed in selection sort. We could have said it was executed n^2 times and still obtained the same big-O complexity.
- Once you get the hang of this, you can quickly zero in on what is relevant for determining asymptotic complexity. For example, you can usually ignore everything that is not in the innermost loop (why?).
- Main difficulty: estimating running time for recursive programs

Analysis of merge-sort:

```
public static Comparable[] mergeSort(Comparable[] A, int low, int high)
 if (low < high - 1) //at least three elements<-- cost = c0, 1 time
   {int mid = (low + high)/2;                   <-- cost = c1, 1 time
    Comparable[] A1 = mergeSort(A,low,mid);     <-- cost = ??, 1 time
    Comparable[] A2 = mergeSort(A,mid+1,high);<-- cost = ??, 1 time
    return merge(A1,A2);}            <-- cost = c2*n + c3 (shown before)
   ....
```

Recurrence equation:

$$T(n) = (c0+c1) + 2T(n/2) + (c2*n + c3)$$

$$T(1) = c4$$

How do we solve this recurrence equation?

```
Recurrence equation:
     T(n) = (c0+c1) + 2T(n/2) + (c2*n + c3)
     T(1) = c4
```

Simplify by dropping lower-order terms:

```
Recurrence equation:
     T(n) = 2T(n/2) + n
     T(1) = 1
```

It can be shown that T(n) = $O(\text{nlog}_2(\text{n}))$.

Remarks

- No general techniques known for solving recurrences (like integration).
- For CS 211, just remember common patterns.
- CS 280: bag of tricks for solving recurrences that arise in practice.

Cheat Sheet for closed-form expressions

Recurrence relation	Closed-form	Example
c(1) = a c(n) = b + c(n-1)	c(n) = O(n)	Linear search
c(1) = a c(n) = b*n + c(n-1)	c(n) = O(n^2)	Quicksort
c(1) = a c(n) = b + c(n/2)	c(n) = O(log(n))	Binary search
c(1) = a c(n) = b*n + c(n/2)	c(n) = O(n)	
c(1) = a c(n) = b + kc(n/k)	c(n) = O(n)	

```
c(1) = a
c(n) = b*n + 2c(n/2)        c(n) = O(n*log(n)) Mergesort

c(1) = a
c(n) = b*n + kc(n/k)        c(n) = O(n*log(n))

c(1) = a
c(2) = b                                       Fibonacci
c(n) = c(n-1)+c(n-2)+ d     c(n) = O(2^n)
```

Analysis of quicksort: tricky

```
public static void quickSort(Comparable[] A, int l, int h) {
   if (l < h)
       {int p = partition(A,l+1,h,A[l]);
        //move pivot into its final resting place;
        Comparable temp = A[p-1];
  A[p-1] = A[l];
  A[l] = temp;
        //make recursive calls
        quickSort(A,l,p-1);
        quickSort(A,p,h);}}
```

Incorrect attempt:

```
c(1) = 1
c(n) = n      +  2c(n/2)
       -         ------
    partition   sorting the two partitioned arrays
```

Why is this wrong?

Remember: big-O is worst-case complexity.

Worst-case for quicksort: one of the partitioned array is empty, and the other has (n-1) elements!

So actual recurrence relation is

```
c(1) = 1
c(n) = n    +  1 +  c(n-1)
       ----    -----------
    partition  sorting the two partitioned arrays
```

It can be shown that c(n) = $O(n^2)$

On the average (not worst-case), quick-sort runs in $n * log_2(n)$ time, which is why it is usually preferred in practice.

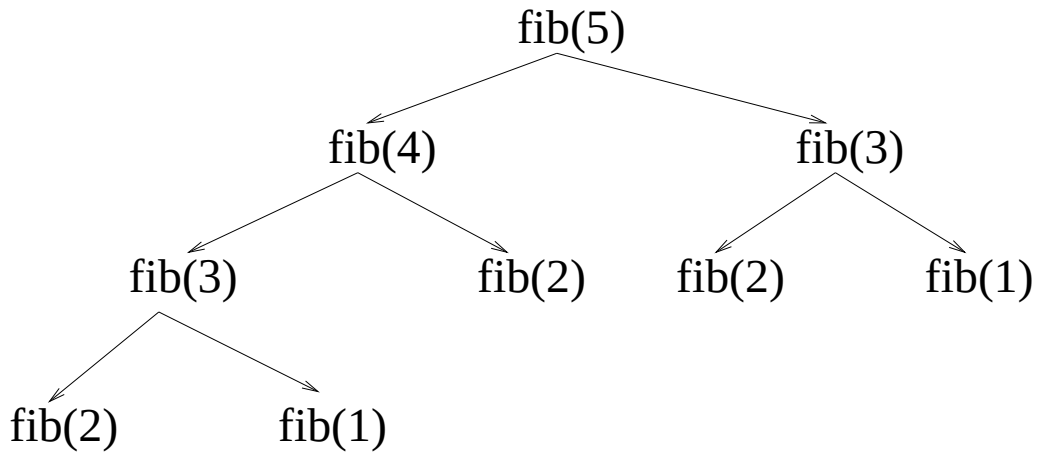
One approach to avoiding worst-case behavior: pick pivot carefully so it partitions array in half. Many heuristics for doing this, but none of them can guarantee that worst-case behavior will not show up.

Programs for the same problem can vary enormously in asymptotic efficiency.

```
fib(n) = fib(n-1) + fib(n-2)
fib(1) = 1
fib(2) = 1
```

Here is a recursive program:

```
static int fib(int n) {
  if (n <= 2) return 1;
  else return fib(n-1) + fib(n-2);
}
```



$$c(n) = c(n-1) + c(n-2) + 2$$

$$c(2) = 1 \quad c(1) = 1$$

For this problem, problem size is n.

It can be shown that $T(n) = O(2^n)$. Cost of computing value is exponential in the size of the input!

Iterative Fibonnacci Code

fib (n) = fib(n-1) + fib(n-2) | n > 2

```
dad = 1
granddad = 1
current = 1;
for (i = 3; i <= n; i++) {
   granddad = dad;
   dad   = current;
   current = dad +granddad;
  }
printf("answer is " + current);
```

Number of times loop is executed is bounded by n.

Each iteration does some constant amount of work.

=> Time complexity of algorithm = O(n).

Summary

1. Asymptotic complexity: measure of space/time required by algorithm
2. Searching array: linear search $O(n)$, binary search $O(log(n))$
3. Sorting array: selection sort $O(n^2)$, merge sort $O(nlog(n))$, quick sort (in-place) $O(n^2)$
4. Matrix operations: matrix-vector product $O(n^2)$, matrix-matrix multiplication $O(n^3)$