Asymptotic Running Time of Algorithms

Asymptotic Complexity: leading term analysis

- Comparing searching and sorting algorithms so far:
  - Count worst-case number of comparisons as function of array size.
  - Drop lower-order terms, floors/ceilings, and constants to come up with asymptotic running time of algorithm.
- We will now generalize this approach to other programs:
  - Count worst-case number of **operations** executed by program as a function of **input size.**
  - **Formalize** definition of big-O complexity to **derive** asymptotic running time of algorithm.

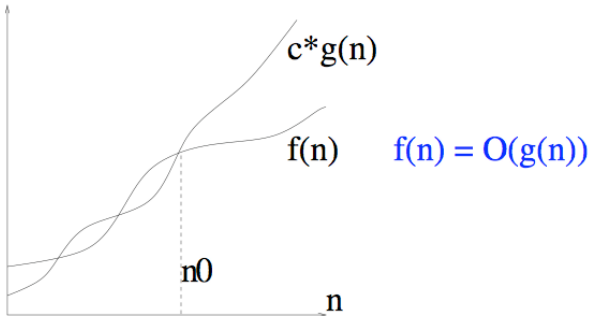
Formal Definition of big-O Notation:

- Let f(n) and g(n) be functions. We say 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, sooner or later g(n) overtakes f(n) as n gets large.
- Example: f(n) = n+5; g(n) = n. Show that f(n) = O(g(n)).
  Choose c = 6:
  $f(n) = n+5 \leq 6*n$ for all $n > 0$.
- Example: $f(n) = 17n$; $g(n) = 3n^2$. 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



- To prove that f(n) = O(g(n)), find an n_0 and c such that $f(n) \leq c * g(n)$ for all $n > n_0$.
- We will call the pair (c, n_0) a *witness pair* for proving that f(n) = O(g(n)).

- For asymptotic complexity, base of logarithms does not matter.
- Let us show that $\log_2(n) = O(\log_b(n))$ for any $b > 1$.
- Need to find a witness pair (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
  $c * \log_b(n) = \log_2(b) * \log_b(n)$
  $= \log_2(b^{\log_b(n)})$
  $= \log_2(n)$
  for all positive n.

Why is Asymptotic Complexity So Important?

- Asymptotic complexity gives an idea of how rapidly the space/time requirements grow as problem size increases.
- Suppose we have a computing device that can execute 1000 complex operations per second. Here is the size problem that can be solved in a second, a minute, and an hour by algorithms of different asymptotic complexity:

Complexity	1 second	1 minute	1 hour
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

What are the basic *OPERATIONS*?

- For searching and sorting algorithms, you can usually determine big-O complexity by counting comparisons.
- Why?
- Usually end up doing some fixed number of arithmetic and/or logical operations per comparison.
- Why don't we count swaps for sorting?
  - e.g., think about selection sort:
    - max of n swaps
    - but must do $O(n^2)$ comparisons
    - swaps and comparisons have similar cost
- Is counting number of comparisons always right?
  - No.
  - e.g., selection sort on linked lists instead of arrays

What *Operations* Should We Count?

- Must do a detailed counting of *all* executed operations.
- Estimate number of primitive operations that RAM model (basic microprocessor) must do to run program:
  - Basic operation: arithmetic/logical operations count as 1 operation.
    - even though adds, multiplies, and divides don't (usually) cost the same
  - Assignment: counts as 1 operation.
    - operation count of righthand side of expression is determined separately.
  - Loop: number of operations per iteration * number of loop iterations.
  - Method invocation: number of operations executed in the invoked method.
    - ignore costs of building stack frames, …
  - Recursion: same as method invocation, but harder to reason about.
  - Ignoring garbage collection, memory hierarchy (caches), …

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:

$= (c1+c2+c6+c7+c8)*n + (c3+c4+c5)*n*(n-1)/2$

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

$= O(n^2)$

Example: 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 = 0;                                  <-- 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 of operations is independent of input size. Just use symbolic constants like c1, c2, … instead.
- Our estimate used a precise count for the number of times the j loop was executed in selection sort (e.g., n*(n-1)/2). Could have said it was executed n^2 times and still have 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
  ....
```

Recurrence equation:

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

$T(1) = c4$ <-- base case

How do we solve this recurrence equation?

Analysis of Merge-Sort

Recurrence equation:

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

$T(1) = c4$

First, simplify by dropping lower-order terms.

Simplified recurrence equation:

$T(n) = 2T(n/2) + n$

$T(1) = 1$

It can be shown that $T(n) = O(n\log(n))$ is a solution to this recurrence.

What do we mean by "*Solution*"

- Recurrence: $T(n) = 2T(n/2) + n$
- Solution: $T(n) = n\log_2 n$

- To prove, substitute $n\log_2 n$ for $T(n)$ in recurrence:

$T(n) = 2T(n/2) + n$

$n\log_2 n = 2(n/2)\log_2(n/2) + n$

$n\log_2 n = n\log_2(n/2) + n$

$n\log_2 n = n(\log_2(n) - \log_2(2)) + n$

$n\log_2 n = n(\log_2(n) - 1) + n$

$n\log_2 n = n\log_2(n) - n + n$

$n\log_2 n = n\log_2 n$

Solving recurrences

- Solving recurrences is like integration --- no general techniques known for solving recurrences.
- For CS 211, we just expect you to remember a few common patterns.
- CS 280, learn a bag of tricks for solving recurrences that arise in practice.

Cheat Sheet for Common Recurrences

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
$c(1) = a$ $c(n) = b*n + c(n/2)$		
$c(1) = a$ $c(n) = b + kc(n/k)$		

Cheat Sheet for Common Recurrences

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)		

Cheat Sheet for Common Recurrences

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)	

Cheat Sheet for Common Recurrences cont.

Recurrence Relation	Closed-Form	Example
c(1) = a c(n) = b*n + 2c(n/2)	c(n) = O(nlog(n))	Mergesort
c(1) = a c(n) = b*n + kc(n/k)	c(n) = O(nlog(n))	
c(1) = a c(2) = b c(n) = c(n-1) + c(n-2) + d	c(n) = $O(2^n)$	Fibonacci

- Don't just memorize these. Try to understand each one.
- When in doubt, guess a solution and see if it works (just like with integration).

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

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:

$c(1) = 1$

$c(n) = n + 2c(n/2)$

n: partition; 2c(n/2): sorting the two partitioned arrays

What is wrong with this analysis?

Analysis of Quicksort: Tricky!

- Remember: big-O is worst-case complexity.
- What is worst-case for Quicksort?
  - one of the partitioned subarrays is empty, and the other subarray has (n-1) elements (n^{th} element is the pivot)!
- So actual worst-case recurrence relation is:

$c(1) = 1$

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

n: partition; c(n-1): sorting partitioned subarrays

- From table, $c(n) = O(n^2)$
- On average (not worst-case) quicksort runs in nlog(n) time.
- One approach to avoiding worst-case behavior: pick pivot carefully so that it always partitions array in half. Many heuristics for doing this, but none of them guarantee worst case will not occur.
- If want to pick pivot as first element, sorted array is worst case. One heuristic is to randomize array order before sorting!

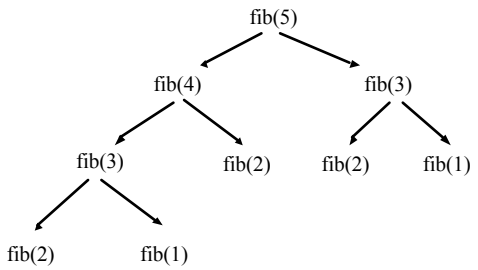
Not all algorithms are created equal

- Programs for same problem can vary enormously in asymptotic efficiency

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

- Here is recursive program for fib(n):

```
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 fib(n) recursively is exponential in size n.

Iterative Code for Fibonnaci

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

```
dad = 1;
grandad = 1;
current = 1;
for (i = 3;  i ≤ 3;  i++) {
        grandad = dad;
        dad = current;
        current = dad + grandad;
}
return (current);
```

- Number of times loop is executed? Less than N.
- Number of computations per loop? Fixed amount of work.
- ==> complexity of iterative algorithm = O(n)
- Much, much, much, much, much, … better than $O(2^n)$!

Summary

- Asymptotic complexity:
  - measure of space/time required by an algorithm
  - measure of *algorithm*, not *problem*
- Searching array:
  - linear search O(n)
  - binary search O(log(n))
- Sorting array:
  - SelectionSort: $O(n^2)$
  - MergeSort: O(nlog(n))
  - Quicksort: $O(n^2)$
    - sorts in place
    - behaves more like O(nlog(n)) in practice
- Matrix operations:
  - Matrix-vector product: $O(n^2)$
  - Matrix-matrix multiplication: $O(n^3)$

Closing Remarks

- Might think that as computers get faster, asymptotic complexity and design of efficient algorithms is less important.
- NOT TRUE!
- As computers get bigger/faster/cheaper, the size of data (N) gets larger
- Moore's Law: ~ ***computers double in speed every 3 years***
- Speed is $O(2^{(years/3)})$.
- If problem size grows at least $O(2^{(years/3)})$, then it's a wash for O(n) algorithms.
- For things worse than O(n) such as $O(n^2\log(n))$, we are rapidly losing computational ground.
- Need more efficient algorithms now more than ever.

	10	50	100	300	1000
5n	50	250	500	1500	5000
nlogn	33	282	665	2469	9966
n^2	100	2500	10,000	90,000	1,000,000
n^3	1000	125,000	1,000,000	27 million	1 billion
2^n	1024	a 16-digit number	a 31-digit number	a 91-digit number	a 302-digit number
n!	3.6 million	a 65-digit number	a 161-digit number	a 623-digit number	unimagin-ably large
n^n	10 billion	an 85-digit number	a 201-digit number	a 744-digit number	unimagin-ably large

- protons in the known universe ~ 126 digits
- □sec since the big bang ~ 24 digits

- Source: D. Harel, *Algorithmics*

How long would it take @ 1 instruction / □sec ?

	10	20	50	100	300
n^2	1/10,000 sec	1/2500 sec	1/400 sec	1/100 sec	9/100 sec
n^5	1/10 sec	3.2 sec	5.2 min	2.8 hr	28.1 days
2^n	1/1000 sec	1 sec	35.7 yr	400 trillion centuries	a 75-digit number of centuries
n^n	2.8 hr	3.3 trillion years	a 70-digit number of centuries	a 185-digit number of centuries	a 728-digit number of centuries

- the big bang was 15 billion years ago ($5*10^{17}$ secs)

- Source: D. Harel, *Algorithmics*

Asymptotic complexity and efficient algorithms

- becomes more important as technology improves
- can handle larger problems

Human genome = 3.5 billion nucleotides ~ 1 Gb
@ 1 base-pair instruction / □sec

- n^2 □ 388445 years
- n log n □ 30.824 hours
- n □ 1 hour