Algorithm analysis

Comp Sci 1575 Data Structures

Missouri S&T Computer Science

Complexity

"Any intelligent fool can make things bigger and more complex. It takes a touch of genius and a lot of courage to move in the opposite direction."
https://en.wikipedia.org/wiki/E._F._Schumacher

Outline

Outline

Evaluating algorithms

How to measure the efficiency of an algorithm?

1. Empirical comparison (run programs) - Problems?
2. Asymptotic algorithm analysis

What impacts the efficiency of an algorithm or data structure?

Evaluating algorithms

What impacts the efficiency of an algorithm or data structure?

- For most algorithms, running time depends on "size" of the input
- For data structures the space depends on the "size" of the input.
- Time cost is expressed as $T(n)$ for some function T on input size n. Draw this.

Outline

Rate of growth?

How does T increase with n?

```
// Return position of largest value
// in "A" of size "n"
int largest(int A[], int n)
{
  int currlarge = 0; // Holds largest element pos

  for(int i = 1; i < n; i++) // For each element
    if(A[currlarge] < A[i]) // if A[i] is larger
      currlarge = i; // remember its position

  return currlarge; // Return largest position
}
```

Define a constant, c, the amount of time required to compare two integers in the above function largest, and thus:
$T(n) = cn$

Draw plot.

Rate of growth?

How does T increase with n?

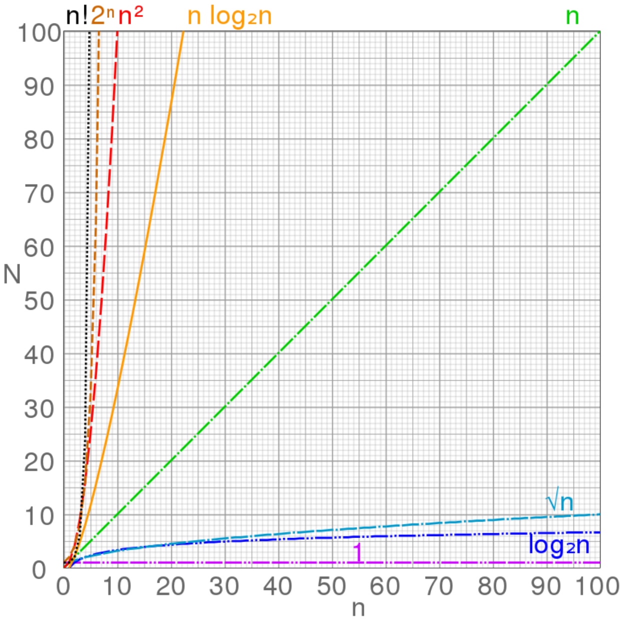
```
sum = 0;

for(i = 1; i <= n; i++)
  for(j = 1; j <= n; j++)
    sum++;
```

We can assume that incrementing takes constant time; call this time c_2, and thus:
$T(n) = c_2 n^2$

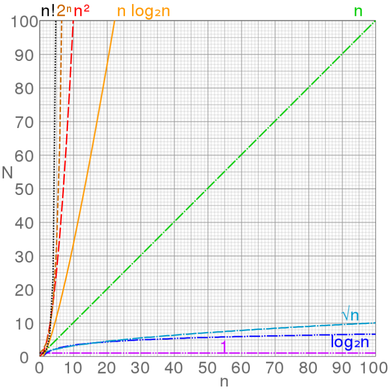
Draw plot.

Rates of growth

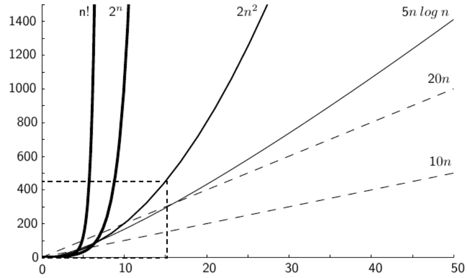


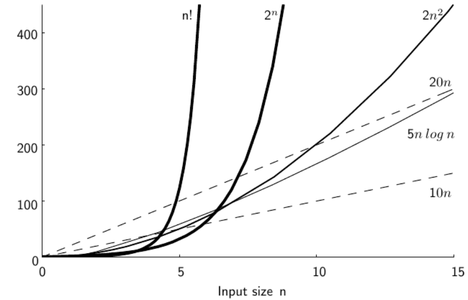
Rates of growth

$O(1)$	constant
$O(\log \log n)$	double log
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential
$O(n!)$	factorial



Rates of growth





Rate of growth?

How does T increase with n?

```
// Return pos of value k in A of size n
int seqSearch(int A[], int n, int k)
{
  for(int i=0; i<n; i++)
    if(A[n] == k)
      return n;

  return -1; // -1 signifies not found
}
```

Constant simple operations plus for() loop: $T(n) = cn$

- Is this always true?
- What if our array is randomly sorted?
- What if our array is fully sorted?
- Whas is out data distribution?

Outline

Best, Worst, Average Cases

Not all inputs of a given size take the same time to run.

Sequential search for K in an array of n integers: Begin at first element in array and look at each element in turn until K is found

- Best case: ?
- Worst case: ?
- Average case: ?

While average time appears to be the fairest measure, it may be difficult to determine; it requires knowledge of the input data distribution.

When is the worst case time important?

Which is best depends on the real world problem being solved!

Outline

Outline

Big-Oh (O) upper bound

- **Definition**: For $T(n)$ a non-negatively valued function, T(n) is in the set O(f(n)) if there exist two positive constants c and n_0 such that $T(n) \leq cf(n)$ for all $n > n_0$.
- **Use**: The algorithm is in $O(n^2)$ in the $\{best, \ average, \ worst\}$ case.
- **Meaning**: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in less than $cf(n)$ steps in $\{best, \ average, \ worst\}$ case.

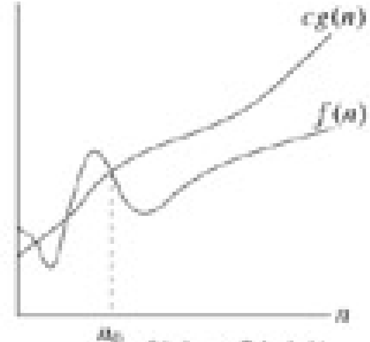
Notation for "is in": $\in$

Big-Oh (O)

Big-oh notation indicates an upper bound.

- Example: If $T(n) = 3n^2$ then $T(n)$ is in $O(n^2)$
- Look for the tightest upper bound:
  While $T(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$.

In image, everywhere to right of n_0 (dashed vertical line) the lower line, $f(n)$, is $\leq$ the top line, $cg(n)$, thus $f(n) \in O(g(n))$:



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

Big-Oh (O) for sequential search


```
// Return pos of value k in A of size n
int seqSearch(int A[], int n, int k)
{
  for(int i = 0; i < n; i++)
    if(A[n] == k)
      return n;

  return -1;
}
```

If visiting and examining one value in the array requires c_s steps where c_s is a positive number, and if the value we search for has equal probability of appearing in any position in the array, then in the average case $T(n) = c_s n/2$. For all values of $n > 1$, $c_s n/2 \leq c_s n$. Therefore, by the definition, $T(n)$ is in $O(n)$ for $n_0 = 1$ and $c = c_s$.

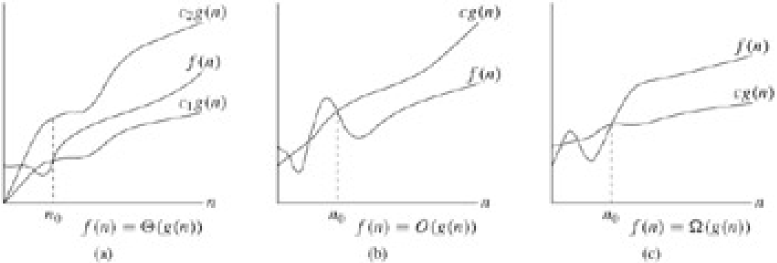
A common mix-up

Big-oh notation indicates an upper bound, and is NOT the same as worst case

- **Big-oh** refers to a bounded **growth rate** as n grows to ∞
- **Best/worst** case is defined for the input of size n that happens to occur among all inputs of size n.

Big-Oh (O)

- $O(g(n)) = \{T(n)$: there exist positive constants c, n_0, such that $0 \leq T(n) \leq cg(n)$ for all $n \geq n_0\}$
- $g(n)$ is an asymptotic upper bound for $T(n)$
- **Middle plot below is Big O**

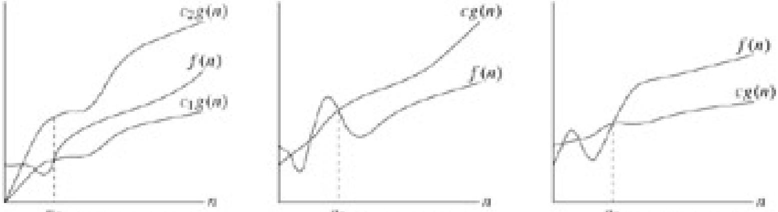


Any values of c?
Growth rate is the important factor.

Outline

Big-Omega (Ω) lower bound

- $\Omega(g(n)) = \{ T(n) :$ there exist positive constants c, n_0, such that $0 \leq cg(n) \leq T(n)$ for all $n \geq n_0 \}$
- $g(n)$ is an asymptotic lower bound for $T(n)$
- **Right plot below is Ω**

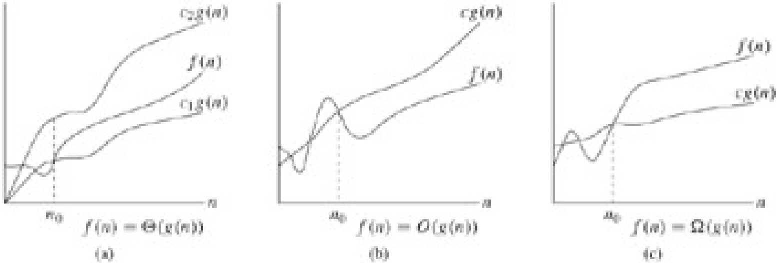


$f(n) = \Theta(g(n))$ (a) $f(n) = O(g(n))$ (b) $f(n) = \Omega(g(n))$ (c)

Outline

Big-Theta (Θ)

- $\Theta(g(n)) = \{T(n)$: there exist positive constants c_1, c_2, n_0, such that $0 \leq c_1 g(n) \leq T(n) \leq c_2 g(n)$ for all $n \geq n_0\}$
- $T(n) = \Theta(g(n))$ if and only if $T(n) \in O(g(n))$ and $T(n) \in \Omega(g(n))$
- $g(n)$ is an asymptotically tight two-sided bound for $T(n)$
- **Left plot below is Θ**



Outline

Little-oh (o)

- $o(g(n)) = \{T(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq T(n) < cg(n)$ for all $n \geq n_0\}$
- $g(n)$ is an upper bound for $T(n)$ that may or may not be asymptotically tight.

Outline

Little-omega (ω)

- $\omega(g(n)) = \{T(n) :$ for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq cg(n) < T(n)$ for all $n \geq n_0\}$
- $g(n)$ is a lower bound for $T(n)$ that is not asymptotically tight

Outline

Outline

Rules to help simplify

- if $A < B$ and $B < C$, then $A < C$
- If $T(n) \in O(f(n))$ and $f(n) \in O(g(n))$, then $T(n) \in O(g(n))$

If some function $f(n)$ is an upper bound for your cost function, then any upper bound for $f(n)$ is also an upper bound for your cost function.

Ignore lower order terms

- Higher-order terms soon swamp the lower-order terms in their contribution to the total cost as n becomes larger.
- For example, if $T(n) = 3n^4 + 5n^2$, then $T(n)$ is in $O(n^4)$.
- The n^2 term contributes relatively little to the total cost for large n.

Why?
Draw this out.

Constants are discarded

- If $T(n) \in O(kf(n))$ for any constant $k < 0$, then $T(n) \in O(f(n))$

You can ignore any multiplicative constants in equations when using big-Oh notation.
Why??

Combinations: sum

- If $T_1(n) \in O(f(n))$ and $T_2(n) \in O(g(n))$, then

$$T_1(n) + T_2(n) \in O(f(n) + g(n)) = O(max(f(n),\ g(n)))$$

Given two parts of a program run in sequence (whether two statements or two sections of code), you need consider only the more expensive part.
Why??

Combinations: product

- If $T_1(n) \in O(f(n))$ and $T_2(n) \in O(g(n))$, then $T_1(n) * T_2(n) \in O(f(n) * g(n))$

If some action is repeated some number of times, and each repetition has the same cost, then the total cost is the cost of the action multiplied by the number of times that the action takes place.

Polynomials

- If $T(n)$ is a polynomial of degree k, then $T(n) = \Theta(n^k)$

Log

- $log^k N \in O(N)$ for any constant k. This tells us that logarithms grow very slowly.

Outline

for() loops

How do we determine the order or growth rate of our code?