UGANDA CHRISTIAN UNIVERSITY

A Center of Excellence in the Heart of Africa

Design and Analysis of Algorithms

Chapter 2: Asymptotic Analysis

Analysis of algorithms

- Measuring efficiency of an algorithm
  - ✓ Time: How long the algorithm takes (running time)
  - ✓ Space: Memory requirement

Time and space

- Time depends on processing speed
  - ✓ Impossible to change for a given hardware
- Space is a function of available memory
  - ✓ Easier to reconfigure and augment.
  - ✓ Typically, we will focus on time, not space

Measuring running time

- Analysis independent of the underlying hardware
  - ✓ Don’t use actual time
  - ✓ Measure in terms of “basic operations”
- Typical basic operations
  - ✓ Compare two values
  - ✓ Assign a value to a variable
- Other operations may be basic, depending on the context
  - ✓ Exchange values of a pair of variables

Input size

- Running time depends on input size
  - ✓ Larger arrays will take longer to sort
- Measure time efficiency as a function of input size
  - ✓ Input size n
  - ✓ Running time t(n)
- Different inputs of size n may each take a different amount of time
- Typically, t(n) is worst-case estimate

Sorting | example 2.1

- Sorting an array with n elements
  - ✓ Naïve algorithms: time proportional to n^2
  - ✓ Best algorithms: time proportional to n log n
- How important is this distinction?
- Typical CPUs process up to 10^8 operations per second
  - ✓ Useful for approximate calculations

1 SAMUEL 7:12

Sorting | example 2.1

- Telephone directory for cell phone users in say China
  - ✓ China has about 1 billion (for easy computation) = 10^9 phones
- Naïve n^2 algorithm requires 10^{18} operations
  - ✓ 10^8 operations per second $\Longrightarrow$ 10^{10} seconds = 2778000 hours = 115700 days = 300 years!
- Smart n log n algorithm takes only about 3 x 10^{10} operations
  - ✓ About 300 seconds, or 5 minutes

Video game | example 2.2

- Several objects on screen
- Basic step: find closest pair of objects
- Given n objects, naïve algorithm is again n^2
  - ✓ For each pair of objects, compute their distance
  - ✓ Report minimum distance over all such pairs
- There is a clever algorithm that takes time n log n

Video game | example 2.2

- The high-resolution monitor has 2500 x 1500 pixels
  - ✓ 3.75 million points
- Suppose we have 500,000 = 5 x 10^5 objects
- The Naïve algorithm takes 25 x 10^{10} steps => 2500 seconds = 42 minutes response time is unacceptable!
- Smart n log n algorithm takes a fraction of a second

Orders of Magnitude

1 SAMUEL 7:12

- When comparing t(n) across problems, focus on orders of magnitude
  - ✓ Ignore constants
- $f(n) = n^3$, eventually grows faster than $g(n) = 5000\ n^2$
  - ✓ For small values of n, f(n) is smaller than g(n)
  - ✓ At n = 5000, f(n) overtakes g(n)
  - ✓ What happens in the limit, as n increases: asymptotic complexity

Typical Functions

- We are interested in orders of magnitude
- Is t(n) proportional to log n, …, n^2, n^3 , …, 2^n?
- Logarithmic, polynomial, exponential …

Typical Functions t(n)...

Input	log n	n	n log n	n^2	n^3	2^n	n!
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7				
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}					

Input size ...

1 SAMUEL 7:12

- How do we fix input size?
- Typically, a natural parameter
  - ✓ For sorting and other problems on arrays: array size
  - ✓ For combinatorial problems: number of objects
  - ✓ For graphs, two parameters: number of vertices and number of edges

Choice of basic operations

- Flexibility in identifying “basic operations”

- Swapping two variables involves three assignments

```
tmp ← x
x ← y
y ← tmp
```

- Number of swaps is 3 times number of assignments

- If we ignore constants, t(n) is of the same order of magnitude even if swapping values is treated as a basic operation

Worst-case complexity

- Running time on input of size n varies across inputs
- Search for K in an unsorted array A

```
i  ← 0
while i < n and A[i] != K do
   i ← i+1
if i < n return i
else return -1
```

Worst-case complexity

- For each n, the worst-case input forces the algorithm to take the maximum amount of time
  - ✓ If K is not in A, the search scans all elements
- Upper bound for the overall running time
  - ✓ Here worst-case is proportional to n for array size n
- Can construct worst-case inputs by examining the algorithm

Average case complexity

- Worst-case may be very rare: pessimistic
- Compute the average time taken over all inputs
- Difficult to compute
  - ✓ Average over what?
  - ✓ Are all inputs equally likely?
  - ✓ Need probability distribution over inputs

Worst-case vs average case

- Worst-case can be unrealistic …
- … but the average case is hard, if not impossible, to compute
- A good worst-case upper bound is useful
- A bad worst-case upper bound may be less informative
  - ✓ Try to "classify" worst-case inputs, look for simpler subclasses

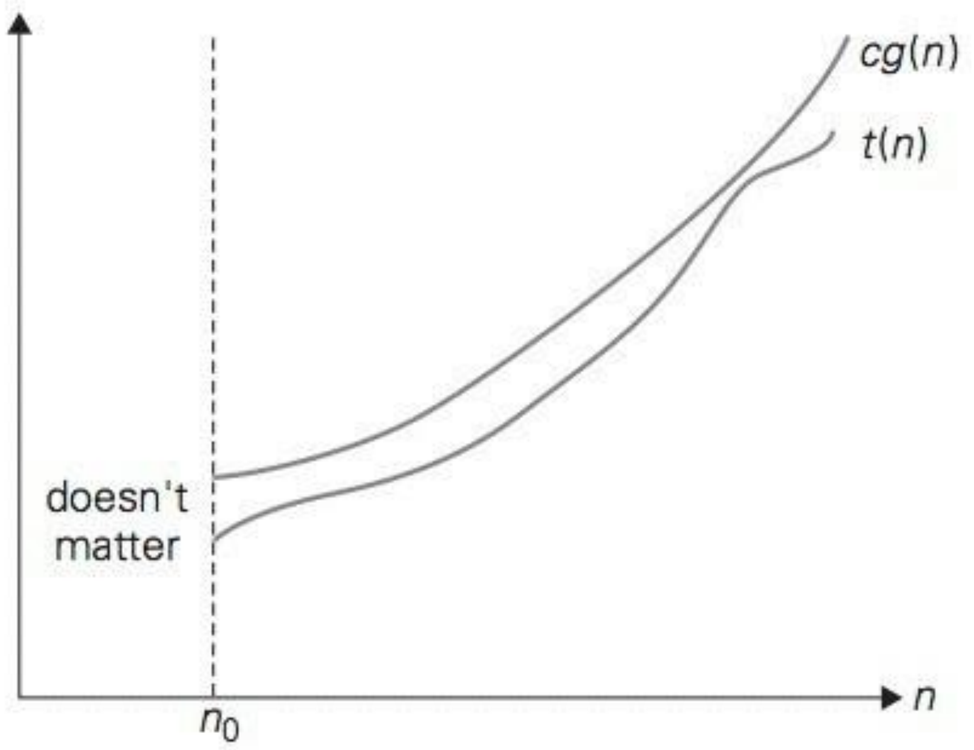
Comparing time efficiency

- We measure time efficiency only up to an order of magnitude
  - ✓ Ignore constants
- How do we compare functions with respect to orders of magnitude?

Upper bounds, “big O”

- t(n) is said to be O(g(n)) if we can find suitable constants c and n_0 so that cg(n) is an upper bound for t(n) for n beyond n_0

- $t(n) \leq cg(n)$ for every $n \geq n_0$



Big O | example

- $100n + 5$ is $O(n)$

  $100n + 5$

  $\leq 100n + 5n$, for $n \geq 1$

  $= 105n \leq 105n$, so $n_0 = 1$, $c = 105$

- n_0 and c are not unique!
- Of course, by the same argument, $100n+5$ is also $O(n)$

Big O | example

- $100n^2 + 20n + 5$ is $O(n^2)$

  $100n^2 + 20n + 5$

  $\leq 100n^2 + 20n^2 + 5n^2$, for $n \geq 1$

  $\leq 125n^2$

  $n_0 = 1$, $c = 125$

- What matters is the highest term

  ✓ $20n + 5$ dominated by $100n^2$

Big O | example

- n^3 is not $O(n^2)$
  - ✓ No matter what c we choose, cn^2 will be dominated by n^3 for $n \geq c$

Useful properties

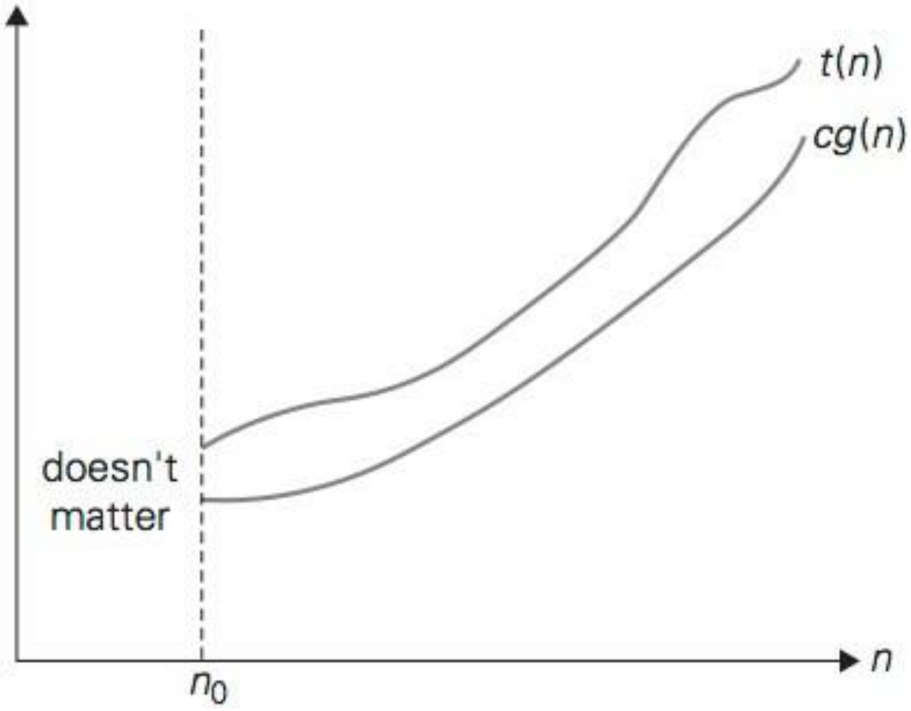
- If
  - ✓ $f_1(n)$ is $O(g_1(n))$
  - ✓ $f_2(n)$ is $O(g_2(n))$
- then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- Proof
  - ✓ $f_1(n) \leq c_1 g_1(n)$ for all $n > n_1$
  - ✓ $f_2(n) \leq c_2 g_2(n)$ for all $n > n_2$

Why is this important?

- Algorithm has two phases
  - ✓ Phase A takes time $O(g_A(n))$
  - ✓ Phase B takes time $O(g_B(n))$
- Algorithm as a whole takes time
  - ✓ $\max(O(g_A(n)),O(g_B(n)))$
- For an algorithm with many phases, least efficient phase is an upper bound for the whole algorithm

Lower bounds, Ω (omega)

- $t(n)$ is said to be $\Omega(g(n))$ if we can find suitable constants c and n_0 so that $cg(n)$ is a lower bound for $t(n)$ for n beyond n_0
  - ✓ $t(n) \geq cg(n)$ for every $n \geq n_0$

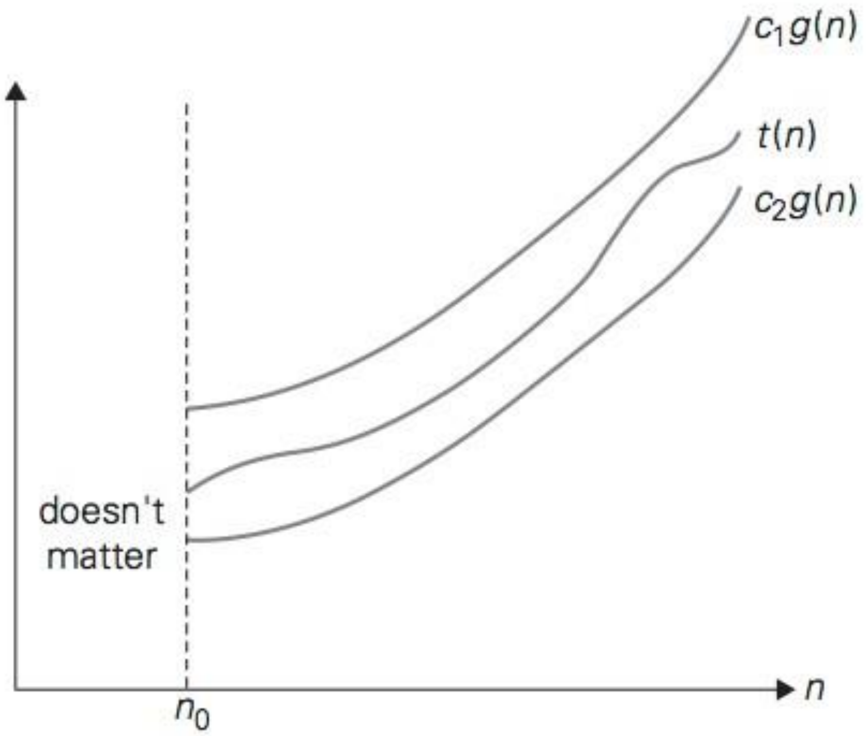


Lower bounds

- n^3 is $\Omega(n^2)$
  - ✓ $n^3 \geq n^2$ for all n
  - ✓ $n_0 = 0$ and $c = 1$
- Typically, we establish lower bounds for problems as a whole, not for individual algorithms
  - Sorting requires $\Omega(n \log n)$ comparisons, no matter how clever the algorithm is.

Tight bounds, Θ (theta)

- $t(n)$ is $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$
- Find suitable constants c_1, c_2, and n_0 so that
  - ✓ $c_2 g(n) \le t(n) \le c_1 g(n)$ for every $n \ge n_0$



Tight bounds

- $n(n-1)/2$ is $\Theta(n^2)$
  - ✓ Upper bound

    $n(n-1)/2 = n^2/2 - n/2 \leq n^2/2$, for $n \geq 0$
  - ✓ Lower bound

    $n(n-1)/2 = n^2/2 - n/2 \geq n^2/2 - (n/2 \times n/2) \geq n^2/4$, for $n \geq 2$
- Choose $n_0 = \max(0,2) = 2$, $c_1 = 1/2$ and $c_2 = 1/4$

Summary

- $f(n) = O(g(n))$ means $g(n)$ is an upper bound for $f(n)$
- Useful to describe the limit of worst-case running time for an algorithm
- $f(n) = \Omega(g(n))$ means $g(n)$ is a lower bound for $f(n)$
- Typically used for classes of problems, not individual algorithms
- $f(n) = \Theta(g(n))$: matching upper and lower bounds Best possible

  algorithm has been found

Calculating Complexity Examples

- Iterative programs
- Recursive programs

Example 2.3

- Maximum value in an array

```
function maxElement(A):
  maxval = A[0]
  for i = 1 to n-1:
      if A[i] > maxval:
          maxval = A[i]
  return(maxval)
```

Example 2.4

- Check if all elements in an array are distinct

```
function noDuplicates(A):
  for i = 0 to n-1:
      for j = i+1 to n-1:
          if A[i] == A[j]:
              return(False)
  return(True)
```

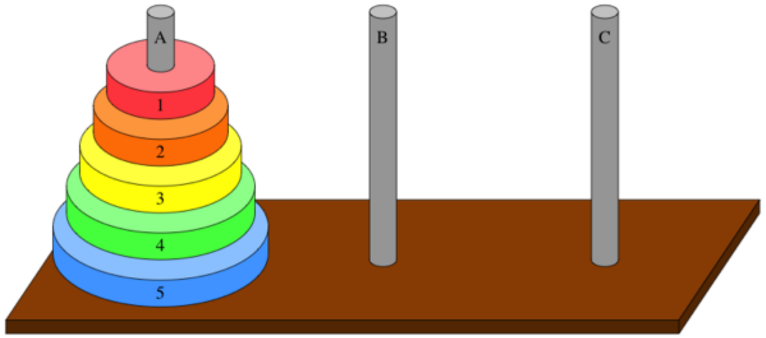
Example 2.5

- Matrix multiplication

```
function matrixMultiply(A,B):
for i = 0 to n-1:
  for j = 0 to n-1:
    C[i][j] = 0
    for k = 0 to n-1:
      C[i][j] = C[i][j] + A[i][k]*B[k][j]
 return(C)
```

Exercise 2.1

- Towers of Hanoi
  - Three pegs, A, B, C
  - Move n disks from A to B
  - Never put a larger disk above a smaller one
  - C is transit peg
- What is the complexity class of this recursive example?



Summary

1 SAMUEL 7:12

- Iterative programs
  - Focus on loops
- Recursive programs
  - Write and solve a recurrence
- Will see more complicated examples
  - Need to be clear about “accounting” for basic operations