Analysis of Algorithms

CS 32 - Data Structures

Jan Michael C. Yap
janmichaelyap@gmail.com

Department of Computer Science
University of the Philippines Diliman

June 19, 2013

Topics

Need for a "unified" programming language

- Remember: data structures are implementations.

Need for a "unified" programming language

- Remember: data structures are implementations.
- Different programming languages, different quirks.

Need for a "unified" programming language

- Remember: data structures are implementations.
- Different programming languages, different quirks.
- Use pseudocodes to abstract the program we want to write (at least ideally, that's what you should be doing, right?)

Need for a "unified" programming language

- Remember: data structures are implementations.
- Different programming languages, different quirks.
- Use pseudocodes to abstract the program we want to write (at least ideally, that's what you should be doing, right?)
- Throughout the course, we will be using EASY to present codes

Topics

EASY operators

EASY operators

- Arithmetic operators: $+, -, \times, /, \hat{}$

EASY operators

- Arithmetic operators: $+, -, \times, /, \hat{}$
- Logical operators: and, or, not

EASY operators

- Arithmetic operators: $+, -, \times, /, \hat{}$
- Logical operators: and, or, not
- Relational operators: $<, \leq, =, \neq, >, \geq, \cdots$

EASY operators

- Arithmetic operators: $+, -, \times, /, \hat{}$
- Logical operators: and, or, not
- Relational operators: $<, \leq, =, \neq, >, \geq, \cdots$
- Miscellaneous mathematical operators: $\lfloor\ \rfloor, \lceil\ \rceil, \mathit{mod}, \log, \cdots$

EASY keywords and syntax

EASY keywords and syntax

- Statement (one line): <statement>

EASY keywords and syntax

- Statement (one line): <statement>
  - $x + 2$

EASY keywords and syntax

- Statement (one line): <statement>
  - $x + 2$
- Successive statement in one line:
  <statement> ; <another statement>

EASY keywords and syntax

- Statement (one line): <statement>
  - $x + 2$
- Successive statement in one line: <statement> ; <another statement>
  - x and y; $z + 1$

EASY keywords and syntax

- Statement (one line): <statement>
  - $x + 2$
- Successive statement in one line:
  <statement> ; <another statement>
  - x and y; $z + 1$
- Assignment statement: <variable name>$\leftarrow$<expression>

EASY keywords and syntax

- Statement (one line): <statement>
  - $x + 2$
- Successive statement in one line:
  <statement> ; <another statement>
  - x and y; $z + 1$
- Assignment statement: <variable name>←<expression>
  - $x \leftarrow 2$

EASY keywords and syntax

- Statement (one line): <statement>
  - $x + 2$
- Successive statement in one line:
  <statement> ; <another statement>
  - x and y; $z + 1$
- Assignment statement: <variable name>←<expression>
  - $x \leftarrow 2$
  - $y \leftarrow x + 2$

EASY keywords and syntax

- Statement (one line): <statement>
  - $x + 2$
- Successive statement in one line:
  <statement> ; <another statement>
  - x and y; $z + 1$
- Assignment statement: <variable name>←<expression>
  - $x \leftarrow 2$
  - $y \leftarrow x + 2$
- array declaration statement:
  array <array name> (<comma sep. array dim. indices>)

EASY keywords and syntax

- Statement (one line): <statement>
  - $x + 2$
- Successive statement in one line:
  <statement> ; <another statement>
  - x and y; $z + 1$
- Assignment statement: <variable name>←<expression>
  - $x \leftarrow 2$
  - $y \leftarrow x + 2$
- array declaration statement:
  array <array name> (<comma sep. array dim. indices>)
  - array A(1:n)

EASY keywords and syntax

- Statement (one line): <statement>
  - $x + 2$
- Successive statement in one line:
  <statement> ; <another statement>
  - x and y; $z + 1$
- Assignment statement: <variable name>←<expression>
  - $x \leftarrow 2$
  - $y \leftarrow x + 2$
- array declaration statement:
  array <array name> (<comma sep. array dim. indices>)
  - array A(1:n)
  - array B(1:m,1:8,1:j)

EASY keywords and syntax

- Statement (one line): <statement>
  - $x + 2$
- Successive statement in one line:
  <statement> ; <another statement>
  - x and y; $z + 1$
- Assignment statement: <variable name>←<expression>
  - $x \leftarrow 2$
  - $y \leftarrow x + 2$
- array declaration statement:
  array <array name> (<comma sep. array dim. indices>)
  - array A(1:n)
  - array B(1:m,1:8,1:j)
- node declaration statement:
  node(<comma separated field name/s>)

EASY keywords and syntax

- go to statement: go to *label*

EASY keywords and syntax

- go to statement: go to *label*
- exit statement: exit

EASY keywords and syntax

- go to statement: go to *label*
- exit statement: exit
- if statement:

```
if <condition 1> then <statement 1>
    else if <condition 2> then <statement 2>
    ...
    else if <condition m - 1> then <statement m - 1>
    else <statement m>
```

EASY keywords and syntax

- case statement:

```
case
  : <condition 1> : <statement 1>
  : <condition 2> : <statement 2>
    ...
  : <condition m - 1> : <statement m - 1>
  :  else : <statement m>
endcase
```

EASY keywords and syntax

- case statement:

```
case
   : <condition 1> : <statement 1>
   : <condition 2> : <statement 2>
     ...
   : <condition m - 1> : <statement m - 1>
   :  else : <statement m>
endcase
```

- while statement:

```
while <condition> do
    <statement/s>
endwhile
```

EASY keywords and syntax

- repeat-until statement:

```
repeat
    <statement/s>
until <condition>
```

EASY keywords and syntax

- repeat-until statement:

```
repeat
    <statement/s>
until <condition>
```

- loop-forever statement:

```
loop
    <statement/s>
forever
```

EASY keywords and syntax

- repeat-until statement:

```
repeat
    <statement/s>
until <condition>
```

- loop-forever statement:

```
loop
    <statement/s>
forever
```

- for statement:

```
for <var name> ← <start index> to <end index> by <step size> do
   <statement/s>
endfor
```

EASY keywords and syntax

- input and output statements:

```
input <list of variable names>
output <list of variable/s or quoted strings>
```

EASY keywords and syntax

- input and output statements:

  ```
  input <list of variable names>
  output <list of variable/s or quoted strings>
  ```

- call statement:
  call <proc name> (<comma separated list of inputs>)

EASY keywords and syntax

- input and output statements:

```
input <list of variable names>
output <list of variable/s or quoted strings>
```

- call statement:
  call <proc name> (<comma separated list of inputs>)
- return statement:

```
return
return(expression)
```

EASY keywords and syntax

- input and output statements:

  ```
  input <list of variable names>
  output <list of variable/s or quoted strings>
  ```

- call statement:
  call <proc name> (<comma separated list of inputs>)
- return statement:

  ```
  return
  return(expression)
  ```

- stop statement: stop

EASY keywords and syntax

- input and output statements:

  ```
  input <list of variable names>
  output <list of variable/s or quoted strings>
  ```

- call statement:
  call <proc name> (<comma separated list of inputs>)
- return statement:

  ```
  return
  return(expression)
  ```

- stop statement: stop
- end statement: end <procedure name>

EASY keywords and syntax

- input and output statements:

  ```
  input <list of variable names>
  output <list of variable/s or quoted strings>
  ```

- call statement:
  call <proc name> (<comma separated list of inputs>)
- return statement:

  ```
  return
  return(expression)
  ```

- stop statement: stop
- end statement: end <procedure name>
- comments: ▷ <comments>

EASY program

EASY program

- procedure structure:

```
procedure <procedure name>(<CS list of inputs>)
  <statement/s>
end <procedure name>
```

EASY program

- procedure structure:

```
procedure <procedure name>(<CS list of inputs>)
  <statement/s>
end <procedure name>
```

  - The MAIN procedure is the starting point of program execution.

EASY program

```
procedure BINARY_SEARCH(A,n,x)
  array A(1:n)
  lower ← 1; upper ← n
  while lower ≤ upper do
    middle ← ⌊ (lower + upper) / 2 ⌋
    case
      : x = A(middle): return(middle)  ▷ successful search
      : x > A(middle): lower ←  middle + 1
      : x < A(middle): upper ←  middle - 1
    endcase
  endwhile
  return(0)  ▷ unsuccessful search
end BINARY_SEARCH
```

EASY program

```
procedure MAIN(A,n,x)
  array T(1:1000)
    ⋮
  input m,q,T(1:m)
    ⋮
  index ← BINARY_SEARCH(T,m,q)
  if index = 0 then output q, " is not found in the array."
    else output q, " is located in array index ", index
    ⋮
end MAIN
```

Topics

Topics

How do we go about analyzing algorithms?

How do we go about analyzing algorithms?

- Algorithms require time and space to perform its task

How do we go about analyzing algorithms?

- Algorithms require time and space to perform its task
  - More particularly, focus is on the rate of growth of the time and space requirement of an algorithm's execution

How do we go about analyzing algorithms?

- Algorithms require time and space to perform its task
  - More particularly, focus is on the rate of growth of the time and space requirement of an algorithm's execution
  - Analysis of algorithms merits a look at their time complexity and space complexity

How do we go about analyzing algorithms?

- Algorithms require time and space to perform its task
  - More particularly, focus is on the rate of growth of the time and space requirement of an algorithm's execution
  - Analysis of algorithms merits a look at their time complexity and space complexity
- But we'd focus more on time complexity analysis
  - We can create some space but not time!

On Time Complexity

On Time Complexity

- The actual/empirical running times of algorithms may actually differ from machine to machine

On Time Complexity

- The actual/empirical running times of algorithms may actually differ from machine to machine
- "Standardize" the specs of the machine

On Time Complexity

- The actual/empirical running times of algorithms may actually differ from machine to machine
- "Standardize" the specs of the machine
  - Single core processor
  - Constant time to execute an instruction/statement
  - Sequential execution of commands

On Time Complexity

On Time Complexity

- Execution of an instruction/statement takes an amount of time, termed the cost of execution (or simply cost)

On Time Complexity

- Execution of an instruction/statement takes an amount of time, termed the cost of execution (or simply cost)
  - Assume declaration statements have zero cost

On Time Complexity

- Execution of an instruction/statement takes an amount of time, termed the cost of execution (or simply cost)
  - Assume declaration statements have zero cost
- Time complexity (or running time) of an algorithm, denoted T(n), is the total cost incurred for its complete execution

On Time Complexity

- Execution of an instruction/statement takes an amount of time, termed the cost of execution (or simply cost)
  - Assume declaration statements have zero cost
- Time complexity (or running time) of an algorithm, denoted T(n), is the total cost incurred for its complete execution
  - Note that the total cost is dependent on the size of the input to the algorithm

On Time Complexity

- Execution of an instruction/statement takes an amount of time, termed the cost of execution (or simply cost)
  - Assume declaration statements have zero cost
- Time complexity (or running time) of an algorithm, denoted T(n), is the total cost incurred for its complete execution
  - Note that the total cost is dependent on the size of the input to the algorithm
  - Time complexity can thus be described as a function of the input size

Analyzing algorithms: a "simple" example

```
procedure INSERTION_SORT(A,n)
  array A(1:n)
  for i←2 to n do
    key ← A(i)
    j ← i - 1
    while j > 0 and A(j) > key do
      A(j + 1) ←  A(j)
      j ← j - 1
    endwhile
    A(j + 1) ←  key
  endfor
end INSERTION_SORT
```

Analyzing algorithms: a "simple" example

```
procedure INSERTION_SORT(A,n)
  array A(1:n)
  for i←2 to n do                          ▷ c1
    key ← A(i)                             ▷ c2
    j ← i - 1                              ▷ c3
    while j > 0 and A(j) > key do          ▷ c4
      A(j + 1) ←  A(j)                     ▷ c5
      j ← j - 1                            ▷ c6
    endwhile
    A(j + 1) ←  key                        ▷ c7
  endfor
end INSERTION_SORT
```

Analyzing algorithms: a "simple" example

```
procedure INSERTION_SORT(A,n)
  array A(1:n)
  for i←2 to n do                          ▷ c1
    key ← A(i)                             ▷ c2
    j ← i - 1                              ▷ c3
    while j > 0 and A(j) > key do          ▷ c4
      A(j + 1) ←  A(j)                     ▷ c5
      j ← j - 1                            ▷ c6
    endwhile
    A(j + 1) ←  key                        ▷ c7
  endfor
end INSERTION_SORT
```

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

More on analyzing algorithms

More on analyzing algorithms

- On top of the input size, the running time of an algorithm also depends on the nature of the input

More on analyzing algorithms

- On top of the input size, the running time of an algorithm also depends on the nature of the input
- Best case analysis: fastest running time possible

More on analyzing algorithms

- On top of the input size, the running time of an algorithm also depends on the nature of the input
- Best case analysis: fastest running time possible
- Average case analysis: most likely running time on average

More on analyzing algorithms

- On top of the input size, the running time of an algorithm also depends on the nature of the input
- Best case analysis: fastest running time possible
- Average case analysis: most likely running time on average
- Worst case analysis: slowest running time possible

On insertion sort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

On insertion sort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

- Best case:
  $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7)$

On insertion sort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

- Best case:
  $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7) \sim an + b$

On insertion sort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

- Best case:
  $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7) \sim an + b$
- Average case:
  $T(n) = \frac{1}{4}(c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{4} - \frac{3c_5}{4} - \frac{3c_6}{4} + c_7)n + (-c_2 - c_3 - c_4 - c_7)$

On insertion sort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

- Best case:
  $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7) \sim an + b$
- Average case:
  $T(n) = \frac{1}{4}(c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{4} - \frac{3c_5}{4} - \frac{3c_6}{4} + c_7)n + (-c_2 - c_3 - c_4 - c_7) \sim an^2 + bn + c$

On insertion sort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

- Best case:
  $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7) \sim an + b$
- Average case:
  $T(n) = \frac{1}{4}(c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{4} - \frac{3c_5}{4} - \frac{3c_6}{4} + c_7)n + (-c_2 - c_3 - c_4 - c_7) \sim an^2 + bn + c$
- Worst case:
  $T(n) = \frac{1}{2}(c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + c_7)n + (-c_2 - c_3 - c_4 - c_7)$

On insertion sort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

- Best case:
  $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7) \sim an + b$
- Average case:
  $T(n) = \frac{1}{4}(c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{4} - \frac{3c_5}{4} - \frac{3c_6}{4} + c_7)n + (-c_2 - c_3 - c_4 - c_7) \sim an^2 + bn + c$
- Worst case:
  $T(n) = \frac{1}{2}(c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + c_7)n + (-c_2 - c_3 - c_4 - c_7) \sim an^2 + bn + c$

Topics

Extending the analysis further

- Recall that time complexity is described as a function of the input size

Extending the analysis further

- Recall that time complexity is described as a function of the input size
- Secondly, we want to focus on the rate of growth of the time complexity of the algorithm

Extending the analysis further

- Recall that time complexity is described as a function of the input size
- Secondly, we want to focus on the rate of growth of the time complexity of the algorithm
- Motivating example: worst case running time of insertion sort, i.e. $T(n) \sim an^2 + bn + c$

Extending the analysis further

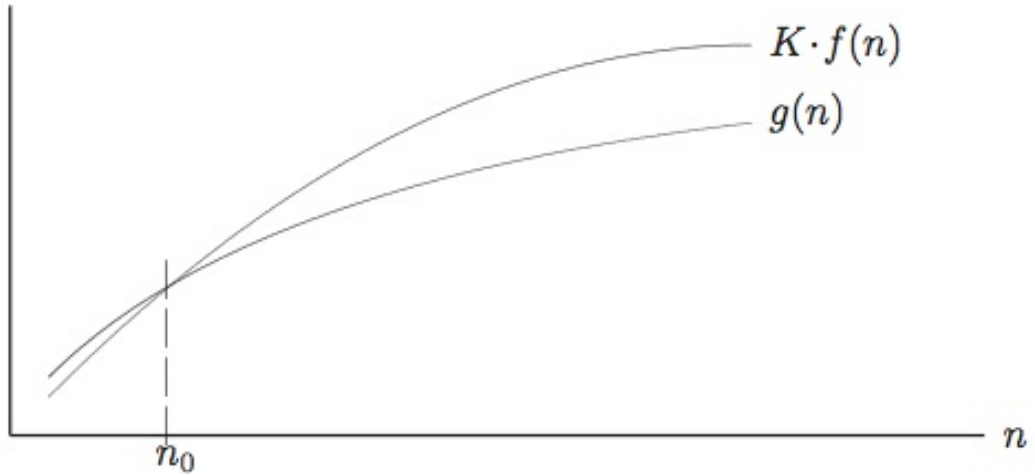
- Recall that time complexity is described as a function of the input size
- Secondly, we want to focus on the rate of growth of the time complexity of the algorithm
- Motivating example: worst case running time of insertion sort, i.e. $T(n) \sim an^2 + bn + c$
  - Note that if we let n be really, REALLY large, the most costly among those terms is the quadratic term!

Extending the analysis further

- Recall that time complexity is described as a function of the input size
- Secondly, we want to focus on the rate of growth of the time complexity of the algorithm
- Motivating example: worst case running time of insertion sort, i.e. $T(n) \sim an^2 + bn + c$
  - Note that if we let n be really, REALLY large, the most costly among those terms is the quadratic term!
- We use asymptotic notations to formally describe the running time of algorithms
  - O-notation, Ω-notation, Θ-notation

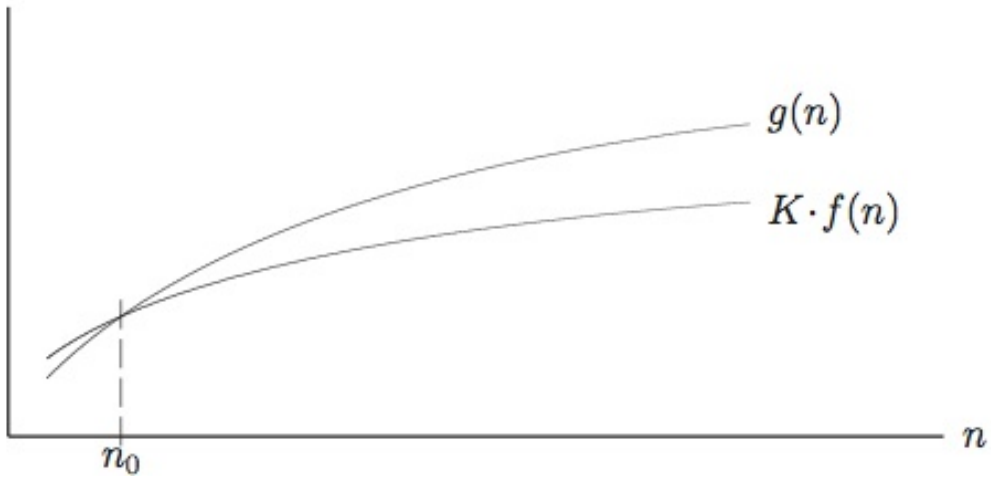
The O-notation

- A function $g(n)$ is $O(f(n))$ (sometimes denoted $g(n) = O(f(n))$ or $g(n) \in O(f(n))$, if there exists two positive constants K and n_0 such that $0 \leq g(n) \leq K \cdot f(n), \forall n \geq n_0$



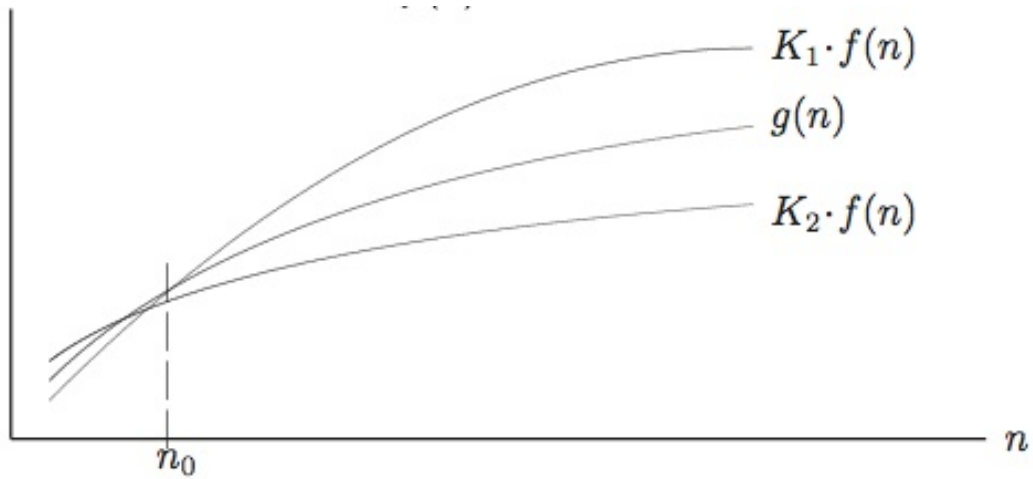
The Ω-notation

- A function $g(n)$ is $\Omega(f(n))$ (sometimes denoted $g(n) = \Omega(f(n))$ or $g(n) \in \Omega(f(n))$, if there exists two positive constants K and n_0 such that $0 \leq K \cdot f(n) \leq g(n), \forall n \geq n_0$



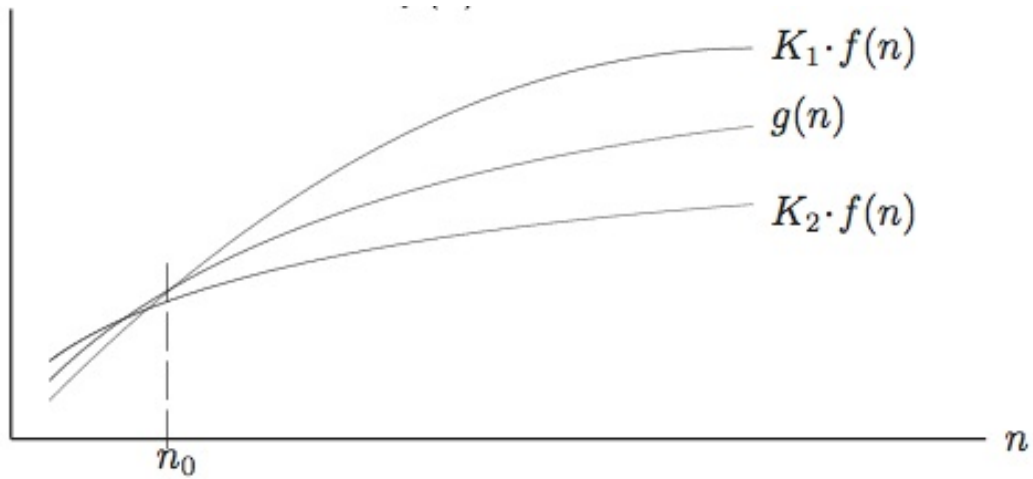
The Θ-notation

- A function $g(n)$ is $\Theta(f(n))$ (sometimes denoted $g(n) = \Theta(f(n))$ or $g(n) \in \Theta(f(n))$, if there exists positive constants K_1, K_2 and n_0 such that $0 \leq K_1 \cdot f(n) \leq g(n) \leq K_2 \cdot f(n), \forall n \geq n_0$



The Θ-notation

- A function $g(n)$ is $\Theta(f(n))$ (sometimes denoted $g(n) = \Theta(f(n))$ or $g(n) \in \Theta(f(n))$, if there exists positive constants K_1, K_2 and n_0 such that $0 \leq K_1 \cdot f(n) \leq g(n) \leq K_2 \cdot f(n), \forall n \geq n_0$



- $g(n) = \Theta(f(n))$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$

Asymptotic notations and time complexity analysis

Asymptotic notations and time complexity analysis

- O-notation for worst case (and average case) time complexity;
  Ω-notation for best case time complexity.

Asymptotic notations and time complexity analysis

- O-notation for worst case (and average case) time complexity; Ω-notation for best case time complexity.
- Θ-notation if the asymptotic worst case and best case running times of the algorithm are the same.

Asymptotic notations and time complexity analysis

- O-notation for worst case (and average case) time complexity; Ω-notation for best case time complexity.
- Θ-notation if the asymptotic worst case and best case running times of the algorithm are the same.
- We"l be doing worst case analysis most of the time, and hence we'll be using the O-notation more except in special cases.

Asymptotic notations and time complexity analysis

- O-notation for worst case (and average case) time complexity; Ω-notation for best case time complexity.
- Θ-notation if the asymptotic worst case and best case running times of the algorithm are the same.
- We"l be doing worst case analysis most of the time, and hence we'll be using the O-notation more except in special cases.
- Some important properties of the O-notation

Asymptotic notations and time complexity analysis

- O-notation for worst case (and average case) time complexity; Ω-notation for best case time complexity.
- Θ-notation if the asymptotic worst case and best case running times of the algorithm are the same.
- We"l be doing worst case analysis most of the time, and hence we'll be using the O-notation more except in special cases.
- Some important properties of the O-notation
  - $f(n) = O(f(n))$

Asymptotic notations and time complexity analysis

- O-notation for worst case (and average case) time complexity; Ω-notation for best case time complexity.
- Θ-notation if the asymptotic worst case and best case running times of the algorithm are the same.
- We"l be doing worst case analysis most of the time, and hence we'll be using the O-notation more except in special cases.
- Some important properties of the O-notation
  - $f(n) = O(f(n))$
  - $c \cdot f(n) = O(f(n))$

Asymptotic notations and time complexity analysis

- O-notation for worst case (and average case) time complexity; Ω-notation for best case time complexity.
- Θ-notation if the asymptotic worst case and best case running times of the algorithm are the same.
- We"ll be doing worst case analysis most of the time, and hence we'll be using the O-notation more except in special cases.
- Some important properties of the O-notation
  - $f(n) = O(f(n))$
  - $c \cdot f(n) = O(f(n))$
  - $O(f(n) + g(n)) = O(\max(f(n), g(n)))$

Asymptotic notations and time complexity analysis

- O-notation for worst case (and average case) time complexity; Ω-notation for best case time complexity.
- Θ-notation if the asymptotic worst case and best case running times of the algorithm are the same.
- We"l be doing worst case analysis most of the time, and hence we'll be using the O-notation more except in special cases.
- Some important properties of the O-notation
  - $f(n) = O(f(n))$
  - $c \cdot f(n) = O(f(n))$
  - $O(f(n) + g(n)) = O(\max(f(n), g(n)))$
  - $O(f(n)) \cdot O((g(n)) = O(f(n) \cdot g(n))$

Topics

Complexity Classes

Complexity class	Common name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	n log n
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$\Omega(2^n)$	Exponential

Assume a 1-billion steps-per-second computer

$f(n)$	$n = 50$	$n = 1000$	$n = 10,000$	$n = 100,000$	$n = 1,000,000$
$\log_2 n$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec
n	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec
$n \log_2 n$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec
n^2	< 1 sec	< 1 sec	< 1 sec	10.00 secs	16.67 min
n^3	< 1 sec	1.00 sec	16.67 min	11.57 days	31.69 yrs
2^n	13.03 days	3.40×10^{284} yrs	6.32×10^{2993} yrs	$3.17 \times 10^{30,086}$ yrs	$3.14 \times 10^{301,013}$ yrs

Topics

Final notes

- In asymptotic time complexity analysis, the bound should be as tight as possible

Final notes

- In asymptotic time complexity analysis, the bound should be as tight as possible
  - If $g(n) = O(f(n))$, f(n) should as "small" as possible with respect to g(n)

Final notes

- In asymptotic time complexity analysis, the bound should be as tight as possible
  - If $g(n) = O(f(n))$, f(n) should as "small" as possible with respect to g(n)
- Although time complexity analysis is of emphasis, space complexity analysis is still of importance.

Final notes

- In asymptotic time complexity analysis, the bound should be as tight as possible
  - If $g(n) = O(f(n))$, f(n) should as "small" as possible with respect to g(n)
- Although time complexity analysis is of emphasis, space complexity analysis is still of importance.
  - Optimization studies and performing under resource (i.e. memory) constrained environments

END OF LESSON 2