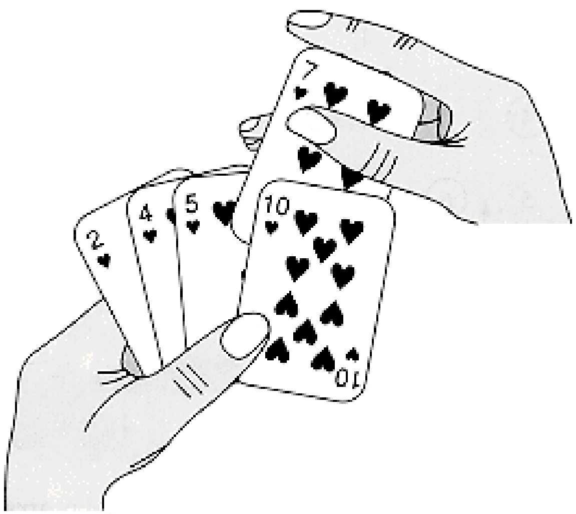
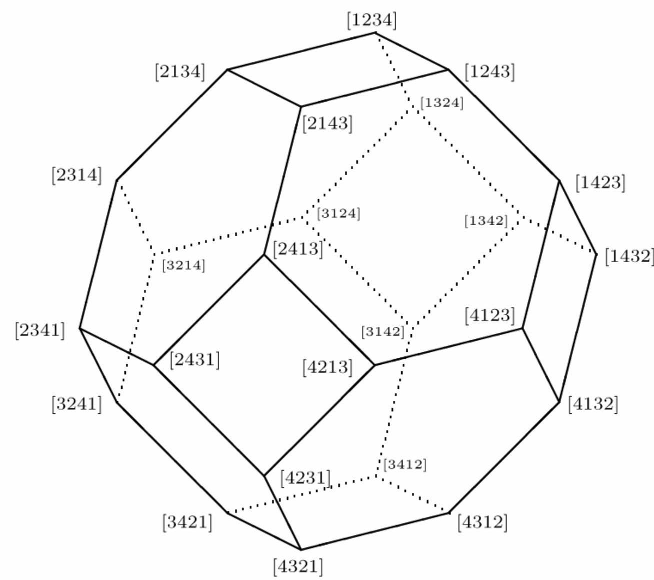
CS3334 Data Structures

Lecture 1: Complexity Analysis





Chee Wei Tan

How to Measure Running Time? (1/2)

- Given a program or procedure, how can you measure its running time?

```
int sum(int n)
{
   int partialSum=0;
   for (int i=1; i<=n; i++)
       partialSum += i*i*i;
   return partialSum;
}
```

- How about using a stop watch and trying all possible values of *n*?

How to Measure Running Time? (2/2)

- No way! Too many *n*'s!
- Do not measure actual running time:
  - Sensitive to program implementation
  - Sensitive to compiler, platform & hardware
- A better way is to use a function to model the running time of a program or procedure.
  - Assume an abstract machine and count the number of "steps" executed by an algorithm

What is a Function?

- In mathematics, a function is a relation between a set of inputs and a set of allowed outputs with the property that each input is related to exactly one output.
- For example, $f(x) = x^3$ (i.e., x is input and $f(x)$ is output)
  - $f(2) = 2^3 = 8$
  - $f(3) = 3^3 = 27$

How to Model Running Time as a Function? (1/2)

- Example: sum of cubes
  (i.e., compute the sum $1^3 + 2^3 + \ldots + n^3$)

```
int sum(int n)
{
    int partialSum=0;
    for (int i=1; i<=n; i++)
        partialSum += i*i*i;
    return partialSum;
}
```

How to Model Running Time as a Function? (2/2)

- Assumption on model:
  - Each simple computer statement takes 1 unit of time

	No. of Execution Times	Unit Cost	Cost
`int partialSum = 0`	1	1	1
`int i=1`	1	1	1
`i<=n`	$n + 1$ (last one for termination)	1	$n + 1$
`i++`	n	1	n
`partialSum += i*i*i`	n	1	n
`return partialSum`	1	1	1
		Total Cost:	$\mathbf{3n + 4}$

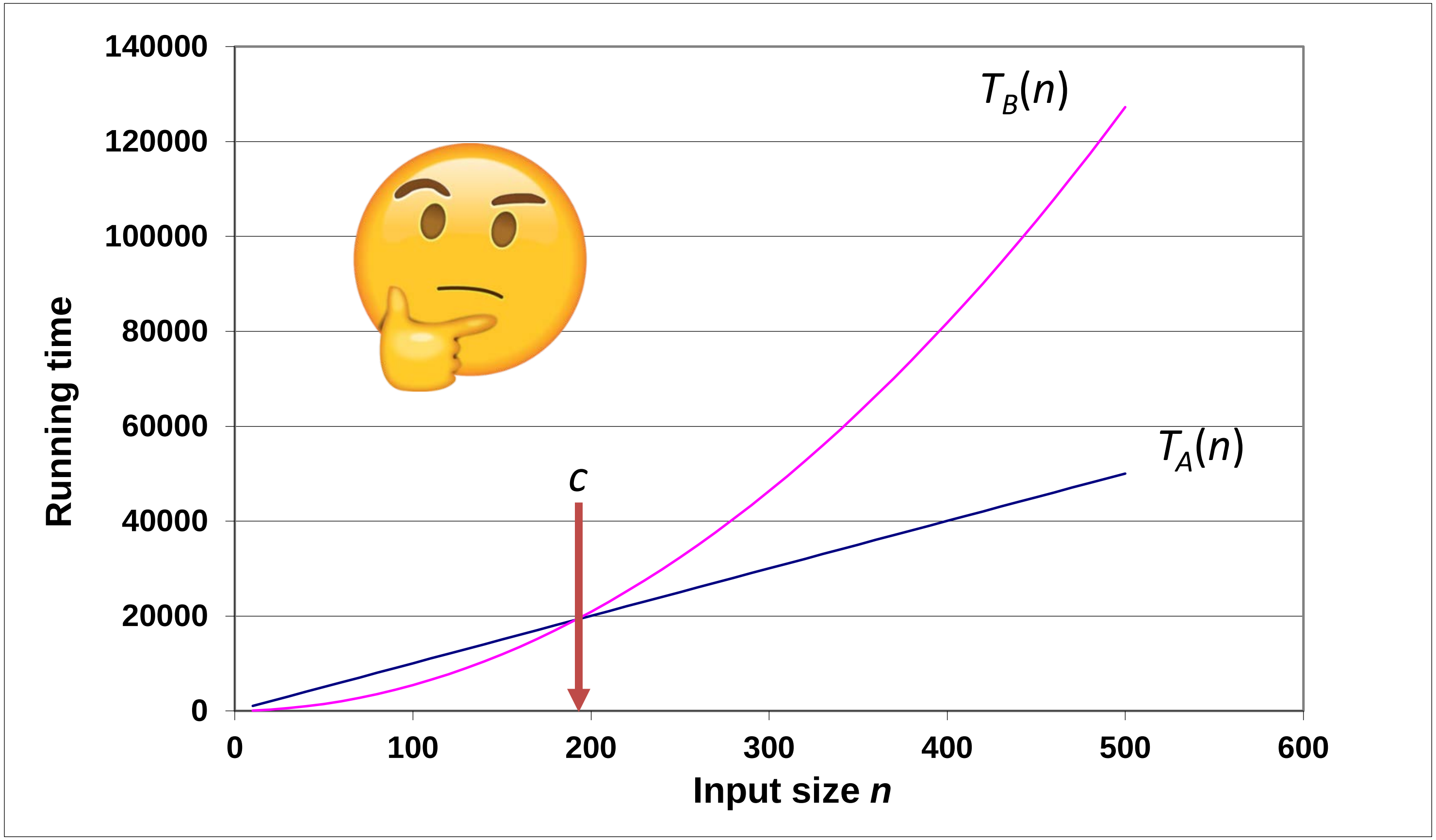
  - Resultant function: $f(n) = 3n + 4$

How to Compare two Functions? (1/4)

- Okay, I know how to model the running time of a program or procedure as a function.
- However, given two algorithms, which one is faster?
- Can I compare the functions of running time models?
  - Example:
    $T_A(n) = 100n + 50$
    $T_B(n) = (0.5)n^2 + 4.5n + 5$

How to Compare two Functions? (2/4)

- $T_A(n) > T_B(n)$ if $n <$ c, but $T_A(n) < T_B(n)$ if $n >$ c

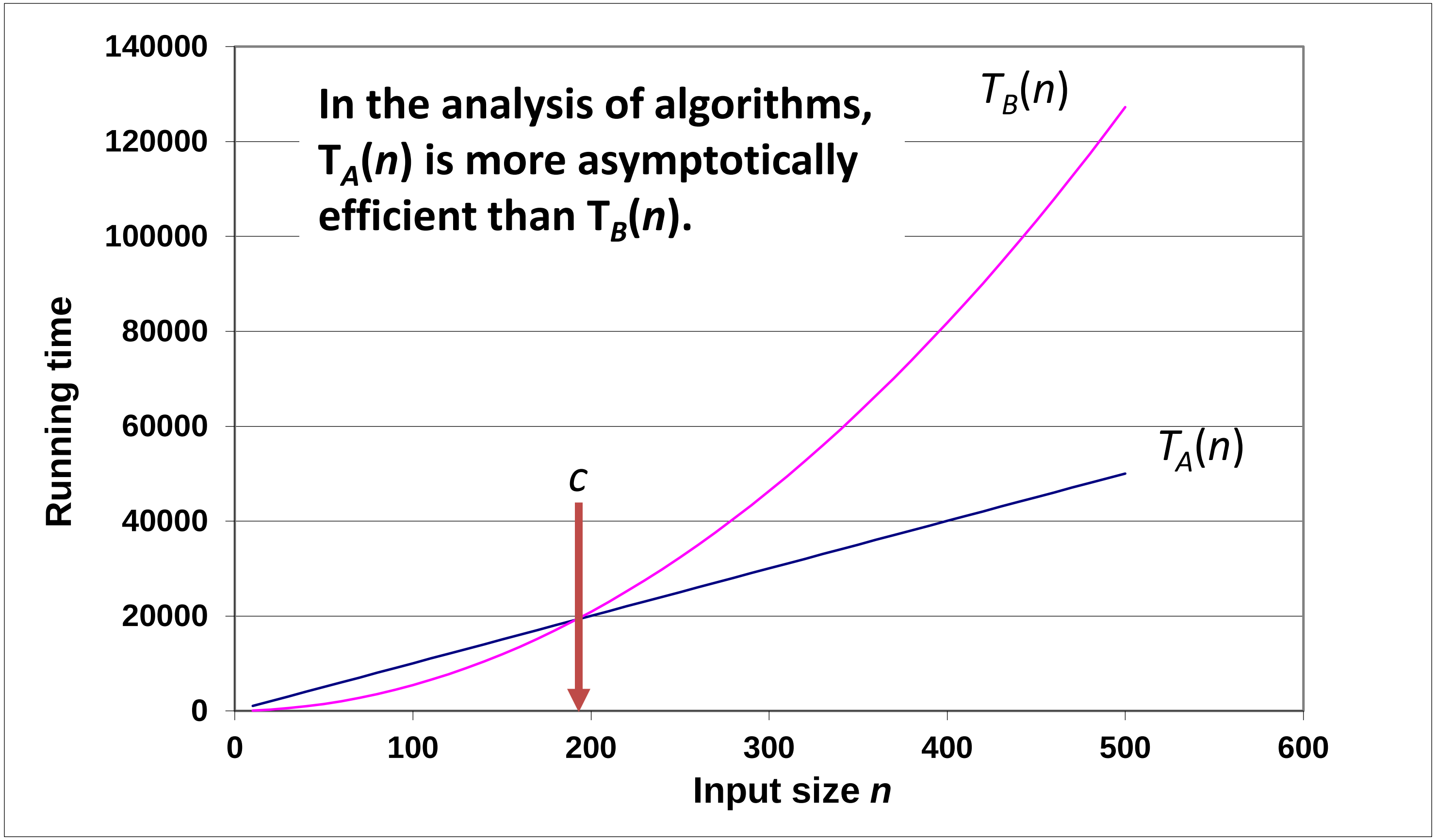


How to Compare two Functions? (3/4)

- We should consider the order of growth of running time, not the actual value!
- The order of growth rate
  – Give a simple characterization of the algorithm's efficiency
  – Allow us to compare the relative performance of alternative algorithms
- In the analysis of algorithms, we just need to consider the performance of algorithms when applied to very very big input datasets, e.g., very very large *n* (i.e., **asymptotic analysis**).

How to Compare two Functions? (4/4)

- $T_A(n) > T_B(n)$ if $n < c$, but $T_A(n) < T_B(n)$ if $n > c$



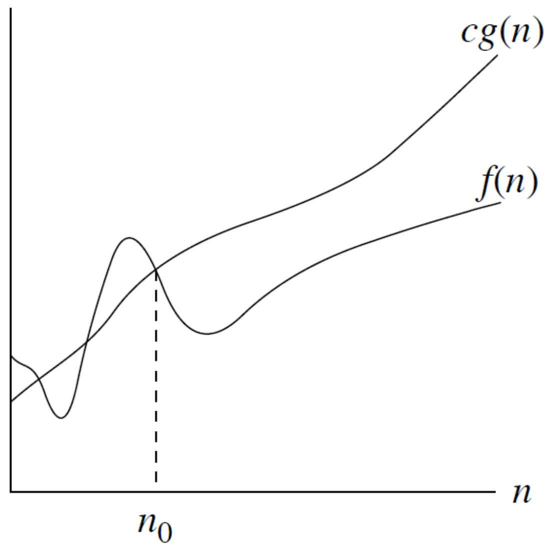
Asymptotic Notation

- How can we indicate running times of algorithms?
  - Need a notation to express the growth rate of a function
  - Learn something professional and new!
- A way to compare "size" of functions:
  - O-notation ("Big-oh") $\approx \leq$ (upper bound)
  - Ω-notation ("Big-omega") $\approx \geq$ (lower bound)
  - Θ-notation ("theta") $\approx =$ (sandwich)

O-notation

- O-notation (“Big-oh”) ≈ ≤ (upper bound)
- We write $f(n) = \mathrm{O}(g(n))$ to
  - Indicate that $f(n)$ is a member of the set $\mathrm{O}(g(n))$
  - Give that $g(n)$ is an upper bound for $f(n)$ to within a constant factor

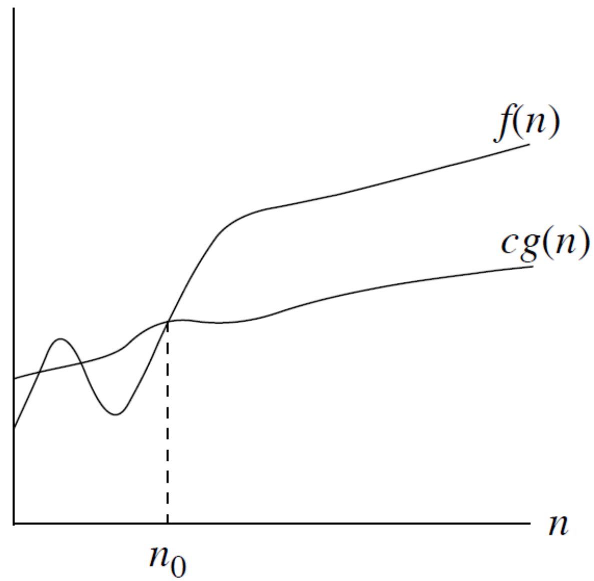
$f(n) = \mathrm{O}(g(n))$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$



Ω-notation

- Ω-notation ("Big-omega") $\approx \geq$ (lower bound)
- We write $f(n) = \Omega(g(n))$ to
  - Indicate that $f(n)$ is a member of the set $\Omega(g(n))$
  - Give that $g(n)$ is a lower bound for $f(n)$ to within a constant factor

$f(n) = \Omega(g(n))$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$

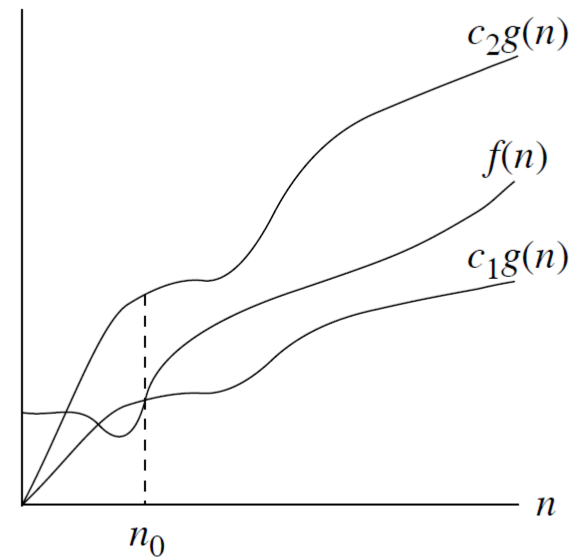


Θ-notation (1/2)

- Θ-notation ("theta") $\approx =$ (sandwich)
- We write $f(n) = \Theta(g(n))$ to
  - Indicate that $f(n)$ is a member of the set $O(g(n))$, give that $g(n)$ is an upper bound for $f(n)$ to within a constant factor
  - Indicate that $f(n)$ is a member of the set $\Omega(g(n))$, give that $g(n)$ is a lower bound for $f(n)$ to within a constant factor
- $f(n) = \Theta(g(n))$ if and only if (1) $f(n) = O(g(n))$ and (2) $f(n) = \Omega(g(n))$

Θ-notation (2/2)

$f(n) = \Theta(g(n))$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$



Find the O-notation of a function (1/3)

- Given a function with **a constant number of terms**, we can obtain the O-notation of the function by using the following three rules:
- **Rule #1:** Sort the terms of the function in decreasing order of their growth rates
  - $\log n$ logarithmic
  - n linear
  - $n\log n$ ---
  - n^2 quadratic
  - n^3 cubic
  - 2^n exponential

Some common complexities in increasing order of their growth rates

Plot the functions and compare

Find the O-notation of a function (2/3)

- **Rule #2:** Ignore lower order terms (i.e., only keep the leading term)
- **Rule #3:** Ignore the coefficient of the leading term
- Note: These three rules DO NOT work for a function with a non-constant number of terms.
  - E.g., Consider an arithmetic progression (AP): $1 + 2 + \ldots + (n\text{-}1) + n$ is not $\mathrm{O}(n)$ but $\mathrm{O}(n^2)$
  - $1 + 2 + \ldots + (n\text{-}1) + n = [(1 + n)n]/2 = \mathrm{n}^2/2 + \mathrm{n}/2 = \mathrm{O}(n^2/2) = \mathrm{O}(n^2)$

Find the O-notation of a function (3/3)

- For example, given $\boldsymbol{f(n) = 5n + 1000\log n + 4n^3}$, we can apply the three rules to find its O-notation:
  1. By **Rule #1**: $f(n) = 4n^3 + 5n + 1000\log n$
  2. By **Rule #2**: $f(n) = O(4n^3)$
  3. By **Rule #3**: $f(n) = O(n^3)$

Other Notes

- How to find the Ω-notation of a function?
  - The same rules can be used to find the Ω-notation of a function.

- Transpose symmetry property
  - $f(n) = \mathrm{O}(g(n))$ if and only if $g(n) = \Omega(f(n))$
  - In other words, if $g(n)$ is the **asymptotic upper bound** of $f(n)$, then $f(n)$ is the **asymptotic lower bound** of $g(n)$, and vice versa.

Time Complexity (1/2)

- For a fixed input size *n*, running time may still depend on the input data (e.g., data distribution and input order)
- **Best case time complexity $T_b(n)$:**
  - The smallest time over all inputs of size *n*
  - Not quite useful
- **Worst case time complexity $T_w(n)$:**
  - The largest time over all inputs of size *n*
  - Give us a performance guarantee that is independent of input data
  - Our focus for most algorithms in this course.

Time Complexity (2/2)

- **Average case time complexity $T_a(n)$:**
  - The average time over all inputs of size n
  - Need an assumption on an input distribution (e.g., in uniform distribution, each possible input is equally likely to happen)
  - Allow us to predict the performance in practice if our assumption on input distribution is realistic
  - More involved mathematically $\Rightarrow$ our focus only for some algorithms

How to Model Running Time as a Function?

- Assumption on model:
  - Each simple computer statement takes 1 unit of time

	No. of Execution Times	Unit Cost	Cost
`int partialSum = 0`	1	1	1
`int i=1`	1	1	1
`i<=n`	$n + 1$ (last one for termination)	1	$n + 1$
`i++`	n	1	n
`partialSum += i*i*i`	n	1	n
`return partialSum`	1	1	1
		Total Cost:	**$3n + 4$**

  - Resultant function: $f(n) = 3n + 4 =$ **$O(n)$**