CSCE 411 Design and Analysis of Algorithms

Asymptotic Notation
(based on Dr. Klappenecker’s lecture slides)
Asymptotic Notations
When estimating the time-complexity of algorithms, we simply want to count the number of operations. We want to be

- independent of the compiler used (esp. about details concerning the number of instructions generated per high-level instructions),
- independent of optimization settings, and architectural details.

This means that performance should only be compared up to multiplication by a constant.

We want to ignore details such as initial filling the pipeline. Therefore, we need to ignore the irregular behavior for small n.
Let $f$, $g$: $\mathbb{N} \to \mathbb{R}$ be functions from the set of natural numbers to the set of real numbers.

Then, $O(g(n))$ is the set of functions

$$O(g(n)) = \{ f(n) \mid \text{there exist a positive constant } c \text{ and a natural number } n_0 \text{ such that } |f(n)| \leq c|g(n)| \text{ for all } n \geq n_0 \}$$

We write $f(n) = O(g(n))$ when $f(n)$ is a member of the set $O(g(n))$.

Example: $2n = O(n^2)$
Example 1: Prove that $2n = O(n^2)$. 
Example 1: Prove that $2n = O(n^2)$.

We can pick $c=1$ and $n_0=2$. Then, $|2n| \leq |n^2|$ for all $n \geq 2$. 
Example 1: Prove that $2n = O(n^2)$.

We can pick $c=1$ and $n_0=2$. Then, $|2n| \leq |n^2|$ for all $n \geq 2$.

Example 2: Does $O(1)$ contain only the constant functions?
Example 1: Prove that $2n = O(n^2)$.

We can pick $c = 1$ and $n_0 = 2$. Then, $|2n| \leq |n^2|$ for all $n \geq 2$.

Example 2: Does $O(1)$ contain only the constant functions?

No. Let’s consider a function $f(n) = 1/n$. We can pick $c = 1$ and $n_0 = 1$. Then, $|1/n| \leq |1|$ for all $n \geq 1$. 
Little Oh

Definition of Big Oh

\[ O(g(n)) = \{ f(n) \mid \text{there exist a positive constant } c \text{ and a natural number } n_0 \text{ such that } |f(n)| \leq c|g(n)| \text{ for all } n \geq n_0 \} \]

Definition of Little Oh

\[ o(g(n)) = \{ f(n) \mid \text{for any positive constant } c > 0, \text{ there exists a natural number } n_0 \text{ such that } |f(n)| < c|g(n)| \text{ for all } n \geq n_0 \} \]

\[ \text{or } \{ f(n) \mid \lim_{n \to \infty} |f(n)|/|g(n)| = 0 \} \]

We write \( f(n) = o(g(n)) \) when \( f(n) \) is a member of the set \( o(g(n)) \).

Example: \( 2n = o(n^2) \)
Example 1: Prove that $2n = o(n^2)$. 
Example 1: Prove that $2n = o(n^2)$.

$$\lim_{n \to \infty} \frac{|2n|}{|n^2|} = 0.$$
Example 1: Prove that $2n = o(n^2)$.

$$\lim_{n \to \infty} \frac{|2n|}{|n^2|} = 0.$$ 

Example 2: $2n^2 = o(n^2)$?
Example 1: Prove that $2n = o(n^2)$.

$$\lim_{n \to \infty} \frac{|2n|}{|n^2|} = 0.$$ 

Example 2: $2n^2 = o(n^2)$?

$$\lim_{n \to \infty} \frac{|2n^2|}{|n^2|} = 2 \neq 0. \text{ Thus, } 2n^2 \neq o(n^2).$$
Let $f, g: \mathbb{N} \to \mathbb{R}$ be functions from the set of natural numbers to the set of real numbers.

Then, $\Omega(g(n))$ is the set of functions

$$\Omega(g(n)) = \{ f(n) \mid \text{there exist a positive constant } c \text{ and a natural number } n_0 \text{ such that } c|g(n)| \leq |f(n)| \text{ for all } n \geq n_0 \}$$

We write $f(n) = \Omega(g(n))$ when $f(n)$ is a member of the set $\Omega(g(n))$.

Example: $2n^3 = \Omega(n^2)$
Let $f, g: \mathbb{N} \rightarrow \mathbb{R}$ be functions from the set of natural numbers to the set of real numbers.

Then, $\Theta(g(n))$ is the set of functions

$$\Theta(g(n)) = \{ f(n) \mid \text{there exist positive constants } c_1, c_2 \text{ and a natural number } n_0 \text{ such that } c_1|f(n)| \leq |f(n)| \leq c_2|g(n)| \text{ for all } n \geq n_0 \}$$

We write $f(n) = \Theta(g(n))$ when $f(n)$ is a member of the set $\Theta(g(n))$.

Example: $2n^2 = \Theta(n^2)$

Thus, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. 
Prove that $\log n! = \Theta(n \log n)$. 
Prove that $\log n! = \Theta(n \log n)$.

Proof.

(i) Prove $\log n! = O(n \log n)$.

$\log n! = \log (1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n)$
$\quad = \log 1 + \log 2 + \ldots + \log n$
$\quad \leq \log n + \log n + \ldots + \log n$
$\quad = n \log n$

Hence, $\log n! = O(n \log n)$. 
Prove that $\log n! = \Theta(n \log n)$.

Proof.

(ii) Prove $\log n! = \Omega(n \log n)$.

$$
\log n! = \log 1 + \log 2 + \cdots + \log n
\geq \log \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) + \cdots + \log n
\geq \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right) \log \left( \left\lfloor \frac{n+1}{2} \right\rfloor \right)
\geq n/2 \log (n/2)
= \Omega(n \log n)
$$

Hence, $\log n! = \Omega(n \log n)$. 
Sorting Lower Bound
How it works:

- incrementally build up longer and longer prefix of the array of keys that is in sorted order
- take the current key, find correct place in sorted prefix, and shift to make room to insert it

Finding the correct place relies on comparing current key to keys in sorted prefix.

Worst-case running time is $\Theta(n^2)$. 
How it works:

- put the keys in a heap data structure
- repeatedly remove the min from the heap

Manipulating the heap involves comparing keys to each other

Worst-case running time is $\Theta(n \log n)$
Mergesort Review

How it works:

- split the array of keys in half
- recursively sort the two halves
- merge the two sorted halves

Merging the two sorted halves involves comparing keys to each other

Worst-case running time is $\Theta(n \log n)$
How it works:

- choose one key to be the pivot
- partition the array of keys into those keys < the pivot and those ≥ the pivot
- recursively sort the two partitions

Partitioning the array involves comparing keys to the pivot

Worst-case running time is $\Theta(n^2)$. 
All these algorithms are comparison-based

- the behavior depends on relative values of keys, not exact values
- behavior on \([1,3,2,4]\) is same as on \([9,25,23,99]\)

Fastest of these algorithms was \(O(n \log n)\).

We will show that is the best you can get with comparison-based sorting.
• Consider any comparison based sorting algorithm
• Represent its behavior on all inputs of a fixed size with a decision tree
• Each tree node corresponds to the execution of a comparison
• Each tree node has two children, depending on whether the parent comparison was true or false
• Each leaf represents correct sorted order for that path
first comparison: check if $a_i \leq a_j$
first comparison: check if $a_i \leq a_j$

Yes

second comparison if $a_i \leq a_j$: check if $a_k \leq a_l$
Decision Tree Diagram

first comparison:
check if \( a_i \leq a_j \)

Yes

second comparison if \( a_i \leq a_j \):
check if \( a_k \leq a_l \)

Yes

third comparison if \( a_i \leq a_j \)
and \( a_k \leq a_l \):
check if \( a_x \leq a_y \)
Decision Tree Diagram

- First comparison: check if $a_i \leq a_j$
  - Yes: second comparison if $a_i \leq a_j$: check if $a_k \leq a_l$
  - No: second comparison if $a_i > a_j$: check if $a_m \leq a_p$
  - Yes: third comparison if $a_i \leq a_j$ and $a_k \leq a_l$: check if $a_x \leq a_y$
Decision Tree Diagram

first comparison: check if $a_i \leq a_j$

second comparison if $a_i \leq a_j$: check if $a_k \leq a_l$

second comparison if $a_i > a_j$: check if $a_m \leq a_p$

Yes

No

Yes

No

Yes

No

Yes

No

...
Example: Insertion Sort for $n = 3$

$a_1 \leq a_2$?
Example: Insertion Sort for $n = 3$

- $a_1 \leq a_2$?
  - Yes
- $a_2 \leq a_3$?
Example: Insertion Sort for $n = 3$

- $a_1 \leq a_2$?
  - Yes

- $a_2 \leq a_3$?
  - Yes

- $a_1 \ a_2 \ a_3$
Example: Insertion Sort for $n = 3$

- **$a_1 \leq a_2$?**
  - Yes
  - **$a_2 \leq a_3$?**
    - No
    - **$a_1 \leq a_3$?**
Example: Insertion Sort for n = 3

- $a_1 \leq a_2$?
  - Yes: $a_2 \leq a_3$?
    - Yes: $a_1 a_3 a_2$
    - No: $a_1 \leq a_3$?
      - Yes: $a_1 a_3 a_2$
      - No: $a_1 a_3 a_2$
Example: Insertion Sort for $n = 3$

- $a_1 \leq a_2$?
  - Yes
  - $a_2 \leq a_3$?
    - Yes
    - No
    - $a_1 \leq a_3$?
      - Yes
      - No
      - $a_1 \leq a_3$?
        - Yes
        - No
        - $a_3, a_1, a_2$
Example: Insertion Sort for $n = 3$

$a_1 \leq a_2$?

No

$a_1 \leq a_3$?
Example: Insertion Sort for $n = 3$

- $a_1 \leq a_2$?
  - No
    - $a_1 \leq a_3$?
      - Yes
        - $a_2 \ a_1 \ a_3$
Example: Insertion Sort for n = 3

- \( a_1 \leq a_2 \)?
  - No

- \( a_1 \leq a_3 \)?
  - No

- \( a_2 \leq a_3 \)?
Example: Insertion Sort for n = 3

\[
\begin{align*}
a_1 \leq a_2? & \quad \text{No} \\
a_1 \leq a_3? & \quad \text{No} \\
a_2 \leq a_3? & \quad \text{No} \\
\text{Yes} & \quad a_2 \ a_3 \ a_1
\end{align*}
\]
Example: Insertion Sort for $n = 3$

- $a_1 \leq a_2$?
  - No

- $a_1 \leq a_3$?
  - No

- $a_2 \leq a_3$?
  - No

- $a_3 \ a_2 \ a_1$
Example: Insertion Sort for $n = 3$

- $a_1 \leq a_2$?
  - Yes: $a_1 a_2 a_3$
  - No: $a_2 a_1 a_3$

- $a_2 \leq a_3$?
  - Yes: $a_1 a_3 a_2$
  - No: $a_3 a_1 a_2$

- $a_1 \leq a_3$?
  - Yes: $a_2 a_3 a_1$
  - No: $a_3 a_2 a_1$
How Many Leaves?

Must be at least one leaf for each permutation of the input

- otherwise there would be a situation that was not correctly sorted

Number of permutations of n keys is n!.

Idea:

Since there must be a lot of leaves, but each decision tree node only has two children, tree cannot be too shallow

- depth of tree is a lower bound on running time
Height of a binary tree with $n!$ leaves is $\Omega(n \log n)$. 
Height of a binary tree with $n!$ leaves is $\Omega(n \log n)$.

Proof.

The maximum number of leaves in a binary tree with height $h$ is $2^h$. 

$h=1 
2^1$ leaves
Height of a binary tree with $n!$ leaves is $\Omega(n \log n)$.

Proof.

The maximum number of leaves in a binary tree with height $h$ is $2^h$. 

$h=1$  
$2^1$ leaves  

$h=2$  
$2^2$ leaves
Height of a binary tree with \( n! \) leaves is \( \Omega(n \log n) \).

Proof.

The maximum number of leaves in a binary tree with height \( h \) is \( 2^h \).
Height of a binary tree with $n!$ leaves is $\Omega(n \log n)$.

Proof.

Let $h$ be the height of decision tree, so it has at most $2^h$ leaves.

The actual number of leaves is $n!$, hence

$$2^h \geq n!$$

$$h \geq \log(n!) = \Omega(n \log n) \quad \text{(Recall: } \log(n!) = \Theta(n \log n))$$
Any binary tree with \( n! \) leaves has height \( \Omega(n \log n) \).

Decision tree for any comparison-based sorting algorithm on \( n \) keys has height \( \Omega(n \log n) \).

Any comparison-based sorting algorithm has at least one execution with \( \Omega(n \log n) \) comparisons.

Any comparison-based sorting algorithm has \( \Omega(n \log n) \) worst-case running time.