Computational Intelligence on Automation Lab @ NCTU

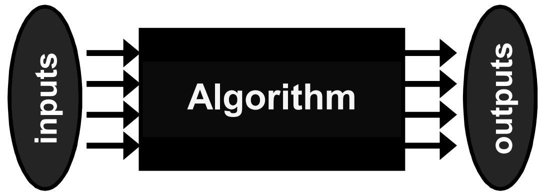
ECM5605(5075) *S'20* Algorithms

Lecture 01:

Fundamentals & Backgrounds: Mathematical Reviews, Insertion Sort, Merge Sort, & Asymptotic Notations

What are Algorithms?

- A well-defined ***computational*** procedure that
  - –takes some value, or set of values, as ***input*** and
  - –produces some value, or set of values, as ***output***;
- A tool to solve a well-specified ***computational problem***.
- Problem vs. Algorithm



Analysis of Algorithms

- The **theoretical study** of computer-program ***performance*** and ***resource usage***
- But what's more important than performance?

✓ **Correctness** ✓ **User-friendliness**
✓ **Functionality** ✓ **Reliability**
✓ **Modularity** ✓ **Extensibility**
✓ **Robustness** ✓ **Programming time**
✓ **Maintainability** ✓ **….**

Basic Issues About Algorithms

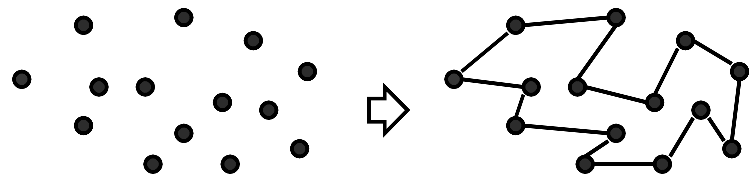
- How to *design* algorithms
- How to *express* algorithms
- Proving *correctness*
- *Efficiency*
  - –*theoretical* analysis
  - –*empirical* analysis
- *Optimality*

Why Study Algorithms?

- Understand ***what can be solved*** and what cannot be solved
  - –Is there any well-defined problem for which we cannot find any algorithm??
- Understand ***how much resource*** including time and space is used to solve this problem
  - –TSP problem: if $n = 20 \Rightarrow$ 20! combinations
  - –Can we find a better algorithm for the same problem?
- Learn how to adapt ***old solutions to new problems***
  - –Many problems seem new but actually not!

Traveling Salesman Problem (TSP)

- **Input** A set of points (cities) $\boldsymbol{P}$ together with a distance $\boldsymbol{d}(\boldsymbol{p},\boldsymbol{q})$ between any pair $\boldsymbol{p},\boldsymbol{q} \in \boldsymbol{P}$
- **Output** The shortest circular route that starts and ends at a given point ($\boldsymbol{s}$) and visits all other points



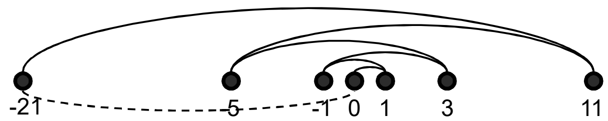
- Exist any correct and efficient algorithm?

Correctness

- For any algorithm, we must prove that it always returns the desired output for all legal instances of the problem.
- For a *correct* TSP tour, check if
  - (1) *Hamiltonian* property: the tour visits *all* points with starting and ending at *the same* point (*tour* or *circuit* property)
  - (2) *Optimality* property: the tour length is the *shortest*
- Algorithm correctness is not obvious in many (optimization) problems.

Nearest Neighbor Tour

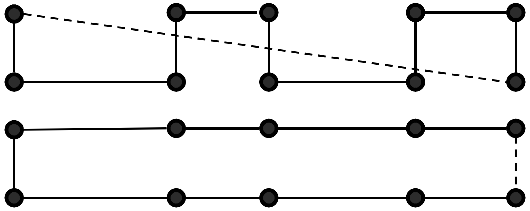
- A popular solution (but wrong!!!)
  - –Start at some point p_0 and then walks to its nearest neighbor p_1 first
  - –Repeat from p_1, etc until done ($p_n \rightarrow p_0$)
- Example



- Starting from the leftmost point will not fix the problem.

Closest-Pair Tour

- Another idea (still wrong!!!)
  - Repeatedly connect the closest pair of points whose connection will not cause a *cycle* or a *three-way* branch until all points are in one tour
- It works on previous example but fails below



A Correct Algorithm

- ***Exhaustive Search***
  - try all possible orderings of the points
  - then select the one which minimizes the total length of the tour
- Since all possible orderings are considered, end up to guarantee the shortest tour
  - total number of permutations: ***n*!** cases
  - *too slow* if *n*>30 (17.9 min@1μsec/case)
- No *efficient*-and-*correct* algorithm exists for TSP so far.

Expressing Algorithms

- Need some way to express the sequence of steps comprising an algorithm
  - Options: English, psuedocode, real programming languages (ex: C/C++, Ada)
- In order of increasing precision
  - English > *pseudocode* > real programs
- Ease of expression
  - English < *pseudocode* < real programs
- Problems need to be carefully specified
  - Ex: "*shortest* tour" is better than "*best* tour"

Mathematical Review (I)

- Ceilings and Floors
  - EX: $\lceil 5/2 \rceil = 3$ $\lfloor 5/2 \rfloor = 2$ $\lceil x/2 \rceil + \lfloor x/2 \rfloor = x$
- Exponentials
  - EX: $(a^m)^n = (a^n)^m = a^{mn}$ and $a^m a^n = a^{m+n}$
- Logarithms
  - EX: $\ln n = \log_e n$ $\log_c(ab) = \log_c a + \log_c b$

    $\lg n = \log_2 n$ $\log_b a^n = n \log_b a$

    $a = b^{\log_b a}$ $a^{\log_b n} = n^{\log_b a}$
- Summation
  - Linearity $\sum_{k=1}^{n}(ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k$

Mathematical Review (II)

- Summations
  - Arithmetic series: Gaussian close form

$$\sum_{i=1}^{n} a_i = \frac{n(a_1 + a_n)}{2}, \text{ if } a_{i+1} = a_i + c\ \forall i > 0$$

  - Geometric series: Geometric close form
  - Harmonic series

Bounding Summations: Technique (1)

- **Proof by induction:**
  1) **Basis**: show formula is true when $n = k$

inductive step

  2) **Hypothesis**: assume formula is true for an arbitrary n
  3) **Step**: show that formula is then true for $n+1$
- **Example**: Gaussian close form
  - Basis: If $n=0$, then $0 = 0(0+1) / 2$
  - Hypothesis: assume $1 + 2 + 3 + \ldots + n = n(n+1) / 2$
  - Step (show true for $n+1$):

$$1 + 2 + \ldots + n + n{+}1 = (1 + 2 + \ldots + n) + (n{+}1)$$
$$= n\,(n{+}1)/2 + n{+}1 = [n(n{+}1) + 2(n{+}1)]/2$$
$$= (n{+}1)(n{+}2)/2 = (n{+}1)(n{+}1 + 1) / 2$$

More on Proof by Induction

- We've been using ***weak induction***
- ***Axiom of induction*** made in 1988

$$\forall \text{predicate } P,\ (P(0) \cap \forall k, [P(k) \Rightarrow P(k+1)]) \Rightarrow \forall n,\ P(n)$$

  - ***predicate***: operator in logic that returns true/false
- Another variation:
  - Basis: show S(0), S(1)
  - Hypothesis: assume S(n) and S(n+1) are true
  - Step: show S(n+2) follows
- ***Strong induction*** implies the procedure
  - Basis: show S(0)
  - Hypothesis: assume S(k) holds for arbitrary $k \leq$ n
  - Step: Show S(n+1) follows

Technique (2): Bounding Terms

- A quick (maybe good) upper bound can be obtained by bounding each term
  - Ex:

$$\sum_{k=1}^{n} k \leq \sum_{k=1}^{n} n = n^2$$

- May give weak bounds using the geometric close form

if $\frac{a_{k+1}}{a_k} \leq r$ for some $\boldsymbol{r} < \mathbf{1}$, then

$$\sum_{k=0}^{n} a_k \leq \sum_{k=0}^{\infty} a_0 r^k = a_0 \sum_{k=0}^{\infty} r^k = a_0 \frac{1}{1-r}$$

Bounding Terms (cont'd)

- Ex: bound the summation $\sum_{k=1}^{\infty} \frac{k}{4^k}$

$$\because \sum_{k=1}^{\infty} \frac{k+1}{4^{k+1}}, \text{the } 1^{st} \text{ term is } \frac{1}{4}, \text{then}$$

$$\frac{(k+2)/4^{k+2}}{(k+1)/4^{k+1}} = \frac{1}{4}\frac{(k+2)}{(k+1)} \le \frac{1}{4}\cdot\frac{2}{1} = \frac{1}{2} < 1$$

$$\therefore \sum_{k=1}^{\infty} \frac{k}{4^k} = \sum_{k=1}^{\infty} \frac{k+1}{4^k} \le \frac{1}{4}\cdot\frac{2}{1-1/2} = \frac{1}{2}$$

- Pitfall example: infinite harmonic series $\sum_{k=1}^{\infty} \frac{1}{k} = ?$

$$\sum_{k=1}^{\infty} \frac{1}{k} = \lim_{n\to\infty} \sum_{k=1}^{n} \frac{1}{k} = \lim_{n\to\infty} (\ln n) = \infty$$

– What's wrong?? $\because \frac{1/k+2}{1/k+1} = 1 \cdot \frac{k+1}{k+2} \approx 1 \text{ when } k \to \infty$

Technique (3): Splitting Summations

- Express the series into the sum of two or more subseries
  - –partition the range of the index
  - –bound each series
- Ex: bound the summation $\sum_{k=0}^{\infty} \frac{k^2}{2^k}$

$$\because \frac{(k+1)^2/2^{k+1}}{k^2/2^k} = \frac{(k+1)^2}{2k^2} \le \frac{8}{9} < 1 \; if k \ge 3$$

$$\therefore \sum_{k=0}^{\infty} \frac{k^2}{2^k} = \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \le \sum_{k=0}^{2} \frac{k^2}{2^k} + \frac{9}{8}\sum_{k=0}^{\infty} (\frac{8}{9})^k$$

Why?

Summations by Parts for H_n

- Harmonic series H_n can be expressed as

$$\underbrace{\frac{1}{1}}_{\text{group 1}} + \underbrace{\frac{1}{2} + \frac{1}{3}}_{\text{group 2}} + \underbrace{\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}}_{\text{group 3}} + \underbrace{\frac{1}{8} + \frac{1}{9} + \frac{1}{10} + \frac{1}{11} + \frac{1}{12} + \frac{1}{13} + \frac{1}{14} + \frac{1}{15}}_{\text{group 4}} + \cdots + \frac{1}{n}$$

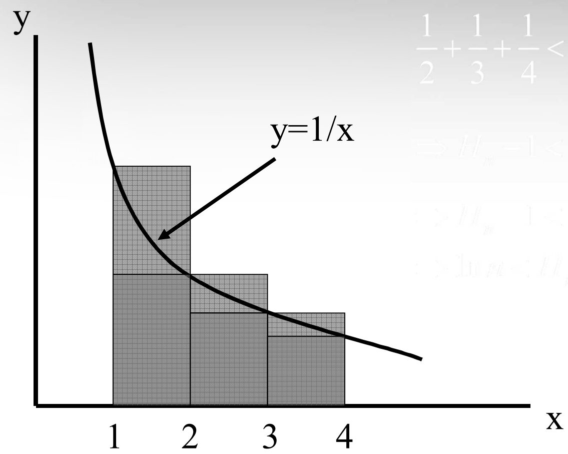
$$sum_{group2} \le \frac{1}{2} + \frac{1}{2}$$

$$sum_{group3} \le \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = 1$$

$$sum_{group4} \le \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} = 1$$

$$\therefore \sum_{k=1}^{n} \frac{1}{k} \le \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i} = \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \le \lg n + 1$$

Technique (4): Approximation by Integrals



$$\frac{1}{2} + \frac{1}{3} + \frac{1}{4} < \int_1^4 \frac{1}{x} < 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4}$$

$$\Rightarrow H_n - 1 < \int_1^n \frac{1}{x} < H_n$$

$$\Rightarrow H_n - 1 < \ln n - \ln 1 < H_n$$

$$\Rightarrow \ln n < H_n < \ln n + 1$$

Get Started: Sorting Problem

- **<u>Input</u>**: a sequence of *n* numbers $\langle a_0, a_1, \ldots, a_{n-1} \rangle$
- **<u>Output</u>**: a permutation $\langle a_{\pi(0)}, a_{\pi(1)}, \ldots, a_{\pi(n-1)} \rangle$ of the input sequence such that

$$a_{\pi(0)} \le a_{\pi(1)} \le \ldots \le a_{\pi(n-1)}$$

  – The numbers to be sorted are known as the ***keys***
  – Permutation: A=<9,6,8> ⇨ A′=<6,8,9>
    ⇨ π(0)=2, π(1)=0, π(2)=1
- Example:
  – Input: 8 2 4 9 3 6 ⇨ Output: 2 3 4 6 8 9
  – π(0)=4 π(1)=0 π(2)=2 π(3)=5 π(4)=1 π(5)=3

Pseudocode of Insertion-Sort()

```
Insertion-Sort(A) //n=length(A[0..n-1])
1 for j ←1 to (length(A)-1) do //A[0] is sorted
2     key ← A[j];
3     // insert A[j] into sorted A[0..j-1]
4     i ← j – 1;
5     while i ≥ 0 and A[i] > key do
6         A[i+1] ← A[i];
7         i ← i – 1;
8     A[i +1] ← key;
```

In-Class Exercise #1: Implement your Insertion-Sort()

Example of Insertion-Sort()

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

2 3 4 6 8 9 ***Done!!***

Features of Insertion-Sort()

- Sorted ***in place***:
  – The numbers are rearranged within the array **A**
  – with at most a *constant* number of them stored outside the array at any time ⇨ *irrelevant* to array length
- ***Loop invariant***:
  – At the start of each iteration of the for loop of line 1-8, the subarray **A**[0..*j*-1] consists of the elements originally in **A**[0..*j*-1] but *in sorted order*

Proving Correctness

- Use loop invariants to prove correctness
  - **Initialization**: true before the 1st iteration
  - **Maintenance**: if is true before an iteration, it remains true before the next iteration
  - **Termination**: when the loop terminates, the *invariants* result in the correctness of the algorithm
- Loop invariants in **Insertion-Sort(A)**
  - **Initialization**: j=1⇨**A**[0] is sorted
  - **Maintenance**: move **A**[j-1], **A**[j-2]… one position to the right until proper **A**[j] position is found
  - **Termination**: when j=n+1⇨**A**[0]…**A**[n] are sorted, the entire array is sorted

Analyze Insertion-Sort()

- Analyzing an algorithm has come to mean ***predicting*** the ***resources*** that the algorithm requires.
  - ***resources***: memory, time, logic gate, communication bandwidth, and etc.
  - ***assumption***: random access machine (***RAM)*** model, which assumes a generic one-processor
    ⇨ instructions are executed one by one and no *concurrent* operations
- Shall have occasion to investigate models for parallel computers and digital hardware

Running Time Analysis

- Depends on the input:
  - an already sorted array is easier to sort
  - Parameterize the running time by *the size of the input* since short sequences are easier to sort than longer ones
- Defined as the number of primitive operations or “*steps*” executed
  - convenient to define the notion of step so that it is as ***machine-independent*** as possible
- Generally, we’re seeking for *upper bound* on the running time because it is a guarantee

Types of Analyses

- **Worst-case**: (usually)
  - $\mathbf{T}(n)$ ≡ *maximum time* of the algorithm on any input of size n
- **Average-case**: (sometimes)
  - $\mathbf{T}(n)$ ≡ *expected time* of the algorithm on all input of size n
  - require assumption of statistical distribution of inputs
- **Best-case**: (bogus)
  - A slow algorithm can cheat and work fast on some input

Exact Analysis of Insertion-Sort()

Insertion-Sort(A) // n=length(**A**[0..n-1])

		cost	times
1	**for** $j \leftarrow 1$ **to** (length(**A**)-1) **do**	c_1	n
2	key $\leftarrow$ **A**[j];	c_2	n-1
3	// insert **A**[j] into sorted **A**[0..j-1]	0	n-1
4	$i \leftarrow j - 1$;	c_4	n-1
5	**while** $i \geq 0$ and **A**[i]>key **do**	c_5	• $\sum_{j=1}^{n-1} t_j$
6	**A**[i+1] $\leftarrow$ **A**[i];	c_6	• $\sum_{j=1}^{n-1} (t_j - 1)$
7	$i \leftarrow i - 1$;	c_7	• $\sum_{j=1}^{n-1} (t_j - 1)$
8	**A**[i +1] $\leftarrow$ key;	c_8	n-1

- The **for** loop is executed (n-1)+1 times (why?)
- t_j: # of times the **while** loop test for value j (i.e., 1+# of elements that have to be slided right to insert the j-th item)
- Step 5 is executed $t_1+t_2+t_3+\ldots+t_{n-1}$ times.
- Step 6 is executed $(t_1-1)+(t_2-1)+\ldots+(t_{n-1}-1)$ times

Exact Analysis (cont'd)

- **Total Running Time $T(n)$:**

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- **Best-case:** If the input is already sorted, all t_j's are 1
  - Linear: $$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$
- **Worst-case**: If array in reverse sorted order, $t_j = j, \forall j$
  - Quadratic:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$
$$= \left(\frac{c_5 + c_6 + c_7}{2}\right)n^2 - \left(c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

Running Time Analysis (revisited)

- Comparison/Analysis depends on computer(s) in use
  - *relative speed (on the same machine)*
    EX: Algorithm A and B run on Machine X
  - *absolute speed (on different machines)*
    EX: Alg A run on Intel Core i7-860 (2.8GHz) vs. Alg B run on AMD FX-9590 (4.7GHz)
- Measure the number of primitive operations or "steps" executed ⇨ ***machine-independent***
  - ignore machine-dependent constants
  - focus on the *growth* of **T**(n) as $n \rightarrow \infty$
  - called "***asymptotic analysis***"

Asymptotic Notation

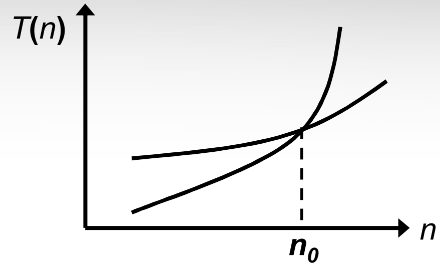
- O notation: asymptotic "*less than/equal to*":
  - f(n)=O(g(n)) implies: f(n) "≤" g(n)
- o notation: asymptotic "*less than*":
  - f(n)=o(g(n)) implies: f(n) "<" g(n)
- Ω notation: asymptotic "*greater than/equal to*":
  - f(n)= Ω(g(n)) implies: f(n) "≥" g(n)
- ω notation: asymptotic "*greater than*":
  - f(n)= Ω(g(n)) implies: f(n) ">" g(n)
- Θ notation: asymptotic "*equal to*":
  - f(n)= Θ(g(n)) implies: f(n) "=" g(n)

Θ-notation

- Mathematical definition:
  A function $f(n)$ is $\Theta(g(n))$ iff
  $\exists$ positive constants c_1, c_2, and n_0
  such that $c_1 g(n) \le f(n) \le c_2 g(n)\ \forall\ n \ge n_0$
- Engineering manipulation:
  – drop lower-order terms
  – ignore leading constants
  EX: $f(n) = 3n^2 + 6n + 202 = \Theta(n^2)$

Comparison of Asymptotic Performance

- When n gets large enough, a $\Theta(n^2)$ algorithm will always beat a $\Theta(n^3)$ algorithm



- However, still shouldn't ignore asymptotic slower algorithms
  – Real-word applications often needs a balance
- Asymptotic analysis helps structure our thinking

Insertion-Sort() (revisited)

- Best-case:
  – $\mathbf{T}(n) = (c_1+c_2+c_4+c_5+c_8)n - (c_2+c_4+c_5+c_8) = \Theta(n)$
- Worst-case:
  – $\mathbf{T}(n) = (c_5/2+c_6/2+c_7/2)n^2 + (c_1+c_2+c_4+c_5/2 - c_6/2 - c_7/2+c_8)n - (c_2+c_4+c_5+c_8) = \Theta(n^2)$
- Average-case: all permutations equally likely
  – $\mathbf{T}(n) = \sum^{n}_{j=2} \Theta(j/2) = \Theta(n^2)$
- When should we use insertion sort?
  – Moderately so for small n
  – Not at all for large n

Summary (Part 1)

- What is Algorithm and its relationship with problem?
- Why do we study Algorithms?
- Review mathematical backgrounds in App. A
- Insertion-Sort()
  – Pseudocode
  – How to prove its correctness
  – Best-case vs. average-case vs. worst-case analysis
- Why do we use asymptotic analysis?
  – Θ-notation
- Up Next ⇨ Merge-Sort() and Recurrence

About Designing Algorithms

- Insertion sort is an *incremental* approach.
  - find the position for one key at one time
- Can we have any other choice?
  - ***Divide-and-Conquer(-and-Combine)***
  - EX: Merge Sort

- Recursive procedure
  - *divide* the problem into sub-problems
  - *conquer* the sub-problem
  - *combine* the results from sub-problems

Pseudocode of Merge-Sort()

Merge-Sort(**A**[0..n-1])

1. If n=0, done
2. *Recursively sort* **A**[0..$\lfloor n/2 \rfloor$] and **A**[$\lfloor n/2 \rfloor$+1..n-1]
3. **Merge**(**A**[0..$\lfloor n/2 \rfloor$], **A**[$\lfloor n/2 \rfloor$+1..n-1)

Key subroutine: Merge

Example of Merge-Sort()

3	8	15	22

1	9	10	14

1	3	8	9	10	14	15	22

Time = $\Theta(n)$ to merge a total of n elements
– linear time

Analyze Merge-Sort()

	time
Merge-Sort(**A**[0..n-1])	**T**(n)
1. If n=1, done	$\Theta(1)$
2. Recursively sort **A**[0..$\lfloor n/2 \rfloor$] and **A**[$\lfloor n/2 \rfloor$ +**1..**n-1] //by **Merge-Sort()**	2**T**(n/2)
3. **Merge**(**A**[0..$\lfloor n/2 \rfloor$], **A**[$\lfloor n/2 \rfloor$ +**1..**n-1])	$\Theta(n)$

- In step 1, $\Theta(1)$ is abusively used
- Step 2 should be **T**($\lceil n/2 \rceil$)+ **T**($\lfloor n/2 \rfloor$) but does not matter in the asymptotic analysis

Recurrence for Merge-Sort()

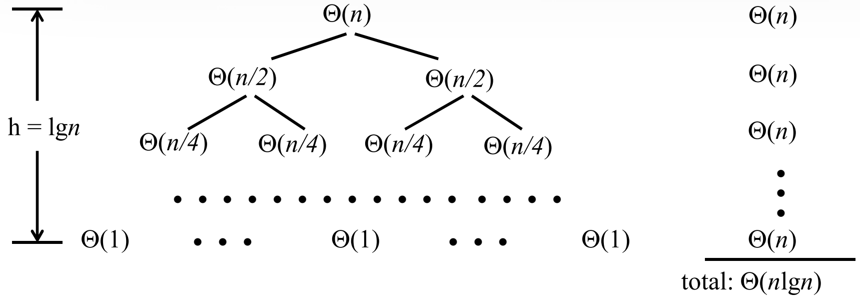
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- Usually omit stating the base case when $\mathbf{T}(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence
- Textbook provides several ways to find a good upper bound on $\mathbf{T}(n)$

Recursion Tree

- Solve $\mathbf{T}(n)=2\mathbf{T}(n/2)+\Theta(n)$ where $c>0$ is constant

  ⇨ $\mathbf{T}(n)=2\mathbf{T}(n/2)+cn$



Merge-Sort() vs Insertion-Sort()

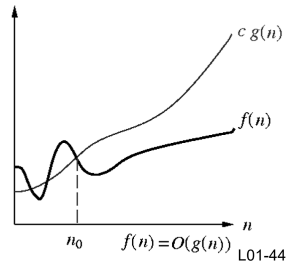
- $\Theta(n\lg n)$ grows more slowly than $\Theta(n^2)$
- Therefore, merge sort asymptotically beats insertion sort in the worst case
- In practice, merge sort beats insertion sort for $n>30$ or so
- We will see the comparison later!

In-Class Exercise #2:

1. Implement your MergeSort()
2. Find *n* for Mergesort() to beat InsertionSort()

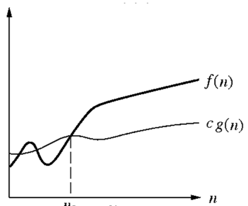
O: Upper Bounding Function

- Def: $f(n)= O(g(n))$ if $\exists$ $c>0$ and $n_0 > 0$ such that $0 \leq \boldsymbol{f(n)} \leq \boldsymbol{cg(n)}$ for all $n \geq n_0$
- Intuition: $f(n)$ "≤" $g(n)$ when we ignore constant multiples and small values of n
- How to show O(Big-Oh) relationships?
  - $f(n) = O(g(n))$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some $c \geq 0$
  - Remember L'Hopitals Rule?
- EX: $2n^2=O(n^3)$
  - $c=1$ and $n_0=2$



Ω : Lower Bounding Function

- Def: $f(n) = O(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that $0 \le \boldsymbol{cg(n)} \le \boldsymbol{f(n)}$ for all $n \ge n_0$
- Intuition: $f(n)$ " $\ge$ " $g(n)$ when we ignore constant multiples and small values of n
- How to show Ω (Big-Omega) relationships?
  - $f(n) = \Omega(g(n))$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some $c \ge 0$
- EX:
  - c=1 and n_0=16



$f(n) = \Omega(g(n))$

Θ: Tightly Bounding Function

- Def: $f(n) = \Theta(g(n))$ if $\exists\ c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \le \boldsymbol{c_1 g(n)} \le \boldsymbol{f(n)} \le \boldsymbol{c_2 g(n)}$ for all $n \ge n_0$
- Intuition: $f(n)$ " $=$ " $g(n)$ when we ignore constant multiples and small values of n
- How to show Θ relationships?
  - Show both "big Oh" ($\mathbf{O}$) and "big Omega" (Ω) relationships
  - $f(n) = \Theta(g(n))$ iff $\lim_{n\to\infty}$ = c for some $c > 0$

o-notation & ω-notation

- **O**-notation and Ω-notation are like ≤ and ≥
- **o**-notation and ω-notation are like < and >
  - $f(n) = o(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that $0 \le \boldsymbol{f(n)} < \boldsymbol{cg(n)}$ for all $n \ge n_0$
  - $f(n) = \omega(g(n))$ if $\exists\ c > 0$ and $n_0 > 0$ such that $0 \le \boldsymbol{cg(n)} < \boldsymbol{f(n)}$ for all $n \ge n_0$
- Example:
  -
  -

Meaning of Asymptotic Notations

- "An algorithm has worst-case run time $\mathbf{O}(f(n))$": there is a constant c s.t. for every n big enough, **every execution** on an input of size n takes **at most** $cf(n)$ time
- "An algorithm has worst-case run time $\Omega(f(n))$": there is a constant c s.t. for every n big enough, **at least one execution** on an input of size n takes **at least** $cf(n)$ time

Asymptotic Properties (I)

- **Transitivity:** If $f(n) = \Pi(g(n))$ and $g(n) = \Pi(h(n))$, then $f(n) = \Pi(h(n))$, where $\Pi = \mathbf{O}, o, \Omega, \omega$, or Θ
- **Rule of sums:** $\Pi(f(n) + g(n)) = \Pi(\max\{f(n), g(n)\})$, where $\Pi = \mathbf{O}, o, \Omega, \omega$, or Θ
- **Rule of sums:** $f(n) + g(n) = \Pi(\max\{f(n), g(n)\})$, where $\Pi = \mathbf{O}, \Omega$, or Θ
- **Rule of products:**

  If $f_1(n) = \Pi(g_1(n))$ and $f_2(n) = \Pi(g_2(n))$,

  then $f_1(n) f_2(n) = \Pi(g_1(n) g_2(n))$,

  where $\Pi = \mathbf{O}, o, \Omega, \omega$, or Θ

Asymptotic Properties (II)

- **Transpose symmetry:**
  - $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
- **Transpose symmetry:**
  - $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$
- **Reflexivity**:
  - $f(n) = \Pi(f(n))$, where $\Pi = \mathbf{O}, \Omega$, or Θ
- **Symmetry:**
  - $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$

Asymptotic Functions

- Polynomial-time complexity:
  - $\mathbf{O}(p(n))$, where n is the **input size** and $p(n)$ is a polynomial function of n $(p(n) = n^{\mathbf{O}(1)})$

1	constant
$\lg^* n$	iterated logarithm
$\lg^{(O(1))} n = \underbrace{\lg \lg \ldots \lg}_{O(1)} n$	–
$\lg n$	logarithmic
$\lg^{O(1)} n = (\lg n)^{O(1)}$	polylogarithmic
$\sqrt{n}$	sublinear
n	linear
$n \lg n$	loglinear
n^2	quadratic
n^3	cubic
n^4	quartic
$2^n, 3^n, \ldots$	exponential
$n!$	factorial
n^n	-

Runtime Comparison

- Run-time comparison: Assume 1000 MIPS, 1 instruction/operation

Order	Θ	$n = 10$	$n = 100$	$n = 10^3$	$n = 10^6$
1	$\Theta(1)$	1×10^{-9} sec	1×10^{-9} sec	1×10^{-9} sec	1×10^{-9} sec
$\lg^* n$	$\Theta(\lg^* n)$	3×10^{-9} sec	3×10^{-9} sec	3×10^{-9} sec	4×10^{-9} sec
$\lg \lg n$	$\Theta(\lg \lg n)$	2×10^{-9} sec	3×10^{-9} sec	3×10^{-9} sec	4×10^{-9} sec
$\lg n$	$\Theta(\lg n)$	3×10^{-9} sec	7×10^{-9} sec	1×10^{-8} sec	2×10^{-8} sec
$\sqrt{n}$	$\Theta(\sqrt{n})$	3×10^{-9} sec	1×10^{-8} sec	3×10^{-8} sec	1×10^{-6} sec
n	$\Theta(n)$	1×10^{-8} sec	1×10^{-7} sec	1×10^{-6} sec	0.001 sec
$n \lg n$	$\Theta(n \lg n)$	3×10^{-8} sec	2×10^{-7} sec	3×10^{-6} sec	0.006 sec
n^2	$\Theta(n^2)$	1×10^{-7} sec	1×10^{-5} sec	0.001 sec	16.7 min
n^3	$\Theta(n^3)$	1×10^{-6} sec	0.001 sec	1 sec	3×10^{5} cent.
2^n	$\Theta(2^n)$	1×10^{-6} sec	3×10^{17} cent.	∞	∞
$n!$	$\Theta(n!)$	0.003 sec	∞	∞	∞

Summary (Part 2)

- Merge Sort
  - Pseudocode
  - Asymptotic analysis by recursion tree
  - Quiz: What are best-case, average-case and worst-case?
- Asymptotic analysis
  - O-notation, Ω-notation and Θ-notation
  - Ordering of asymptotic functions
- Next lecture
  - Recurrence and proving skills
  - Heap sort, Quick sort and other linear sorts