Today’s Lecture

- The structure of the course.
- Why study algorithms.
- How to measure the efficiency of an algorithm.
- Asymptotics (in theory of algorithms).
The Course

Look at the handouts!

Why study algorithms?

- Bag of tricks:
  - Sorting.
  - Data Structures: queues/stacks/heaps/trees.
  - Search: finding proverbial “needle in the haystack”

- Methodology (how to design algorithms):
  - Divide & Conquer.
  - Recursive methods.
  - Randomized method.
  - Dynamic programming

- Useful abstractions:
  - Scheduling classes to Graphs.
  - Job assignment to Matching problem in Graphs.
How to Compare Algorithms?

- Code and Run (experiments):
  - Inputs?
  - Parameters?
  - Bad Implementation!

- Average Case:
  - What is the average input?

- Worse Case:
  - Asymptotics.
  - Analytical Dependency among Components.

Example:

- **Insertion sort:**
  - For \( j=2 \) to \( n \)
    - key=A(\( j \))
    - \( i=j-1 \)
    - while \( i>0 \) and A(\( i \))>key
      - A(i+1)=A(i)
      - A(i)=key
      - \( i-- \)
  - end
  - end

- **Execution:**
  - A: 7 3 5 8 1 2
    - 3 7
    - 3 5 7
    - 3 5 7 8
    - 1 3 5 7 8
    - 1 2 3 5 7 8
About pseudo-code

- What is good about It:
  - It is not really a program just an outline.
  - Enough details to establish the running time.
  - No error-handling mechanisms.

- What is bad about it:
  - It is complicated, i.e.

  for a trivial algorithms it obscures what is really going on...

Analysis

- Running time:
  - Depends on input size & input properties

- Want an upper bound on:
  - Worst case: Max $T(n)$, any input size.
  - Expected: $E[T(n)]$, input taken from a distribution (which?)
    - Example: Sorting arriving TCP/IP packets (they are mostly sorted already).
  - Best Case: Can be used to argue that the algorithm is really bad.
    - Any algorithm can be rewritten to have an excellent “best case” performance.
Example:

- **Insertion sort:**
  
  For $j=2$ to $n \rightarrow n$ times
  
  key=A($j$) \rightarrow n-1 times
  
  $i=j-1$ \rightarrow n-1 times
  
  while $i>0$ and A($i$)>key \rightarrow n-1 times
  
  $A(i+1)=A(i)$
  
  $A(i)=key$
  
  $i--$
  
  end
  
  end

Analysis

- Assume each operation takes 1 time unit (approximation):
  
  $T(n) = n + (n-1) + (n-1) + (n-1) + 3 \sum_{j=2}^{n} (t_j - 1)$

- $t_j$ is in the worst case $j$:
  
  $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$

- Would like to formalize this statement!

- Best running time:
  
  - Outer loop: always executed,
  
  - Inner loop: not executed if Key $\geq$ A($j$), meaning that A is already sorted.
Formalization

- How to formalize that $n(n+1)/2$ was the main term?
- The answer is an asymptotic analysis:
  - Ignore machine-dependent constants.
  - Look at growth of $T(n)$ as $n$ approaches infinity.
- Intuition: drop low-order terms:

$$5n^4 + 10n^2 - 3n + 2 = \Theta(n^4)$$

Ideas: as $n \to \infty$, $\Theta(n^2)$ grows slower than $\Theta(n^4)$.

Example:

- In Insertion Sort, the inner loop was $\Theta(j)$

$$T(n) \approx \sum_{j=2}^{n} \Theta(j) \approx \Theta\left(\sum_{j=2}^{n} j\right) \approx \Theta(n^2)$$

- Is this formal?
  - $\Theta(1) + \Theta(1) = \Theta(1)$
  - Seems to imply:

$$\sum_{j=1}^{n} \Theta(1) \approx \Theta(1) \iff \text{incorrect}$$
Asymptotics

- Big-Oh notation:
  \[ f(n) = O(g(n)) \iff \exists \text{ constant } c, n_0 \text{ s.t. } \forall n \geq n_0 : 0 \leq f(n) \leq cg(n) \]
- Example: \(2n^2 = O(n^6)\) but not vice versa!
- “=” is not equality, but membership in a set.
- Set notation is cumbersome:
  \[ O(g(n)) = \{ f(n) \mid \exists \text{ constant } c, n_0 \text{ s.t. } \forall n \geq n_0 : 0 \leq f(n) \leq cg(n) \} \]
- What does it mean to say \(f(n) = O(n) + n^2\)

Asymptotics

- Small-oh notation
  \[ f(n) = o(g(n)) \iff \forall \text{ constant } c, \exists n_0 \text{ s.t. } \forall n \geq n_0 : 0 \leq f(n) < cg(n) \]
- Example:
  - Prove that \(n = o(n^2)\):
  - Given \(c\) any constant > 0, choose \(n_0 = 2/c\). Then we have:
    \[
    \text{for } n \geq n_0, n^2 \geq \frac{2}{c} n \Rightarrow cn^2 \geq c \left( \frac{2}{c} n \right) > n.
    \]
Omega Notation

- **Big Omega:**

  \[ f(n) = \Omega(g(n)) \iff \exists \text{ constant } c, n_0 \text{ s.t. } \forall n \geq n_0 : 0 \leq cg(n) \leq f(n) \]

- **Small-omega**

  \[ f(n) = \omega(g(n)) \iff \forall \text{ constant } c, \exists n_0 \text{ s.t. } \forall n \geq n_0 : 0 \leq cg(n) < f(n) \]

Transivity

- **Most rules apply:**
  - Example: transitivity
    \[ a \leq b \& b \leq c \implies a \leq c \]
  - Proof:
    \[ f = O(g) \& g = O(h) \implies f = O(h) \]
    \[ f(n) = O(g(n)) \iff \exists \text{ constant } c, n_0 \text{ s.t. } \forall n \geq n_0 : 0 \leq f(n) \leq cg(n) \]
    \[ g(n) = O(h(n)) \iff \exists \text{ constant } c', n_0' \text{ s.t. } \forall n \geq n_0' : 0 \leq g(n) \leq c'h(n) \]
    Take \( n_0'' = \max(n_0, n_0') \), and \( c'' = cc' \). Then
    \[ \forall n \geq n_0'' : 0 \leq f(n) \leq cg(n) \leq cc'h(n) = c''h(n) \]
    \[ \implies f(n) = O(g(n)) \]
Exception

- Not all rules apply:

\[ \forall a, b : \text{Either } a \geq b \text{ or } b \geq a \]

- Is not true for \( O(\ ) \) notation (why?).

---

Theta Notation

- Theta:

\[ f(n) = \Theta(g(n)) \iff \exists \text{ cons. } c_1, c_2, n_0 \text{ s.t. } \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \]

- Often mistaken for Big-Oh notation.

- Example: \[ \frac{n^2}{2} - 2n = \Theta(n^2) \]

- Proof:

  Take \( n_0 = 8 \), then for all \( n \geq n_0 \) we have:

  \[ n^2 / 2 - 2n \geq n^2 / 4 + 8n / 4 - 2n = n^2 / 4 \]

  On the other hand \( n^2 / 2 - 2n < n^2 / 2 \).

  Thus: \( n^2 / 4 \leq n^2 / 2 - 2n \leq n^2 / 2 \), i.e. \( c_1 = 1 / 4, c_2 = 1 / 2 \).
Simple Theorem

- **Claim:** If $f(n) = O(g(n))$ and $g(n) = O(f(n))$ then $f(n) = \Theta(g(n))$

- **Proof:**
  
  $\exists n_1, c_1, \forall n \geq n_1 : 0 \leq f(n) \leq c_1 g(n)$
  
  $\exists n_2, c_2, \forall n \geq n_2 : 0 \leq g(n) \leq c_1 f(n)$
  
  $\Rightarrow \forall n \geq \max(n_1, n_2) : 0 \leq \frac{1}{c_2} g(n) \leq f(n) \leq c_1 g(n)$

Conclusion

- Do we need all these to write program
  - NO!

- Then why we need to know these stuff?
  - It will increase the I.Q.
  - It will make you a better and more conscious programmer.
  - You can say more than, Oh well, the algorithms works!