Introduction
Algorithms

• Goal: map inputs to outputs
  ▪ The mapping is usually defined by a “problem”
  ▪ No “information” is generated… data is “processed”

• Correctness is critical
  ▪ Should prove that the mapping will (almost?) always be performed correctly by your algorithm

• Efficiency is very important
  ▪ What does “efficient” mean? What is being measured?
  ▪ Running time, Space (memory), other resources…
  ▪ Tradeoff: Efficiency vs. ease of design and elegance of implementation
Example Problem: Sorting

• Input is a sequence of n items \( (a_1, a_2, \ldots, a_n) \)
• The mapping we want is determined by a “comparison” operation, denoted by \( \leq \)
• Output is a sequence \( (b_1, b_2, \ldots, b_n) \) such that:
  - \( \{ a_1, a_2, \ldots, a_n \} = \{ b_1, b_2, \ldots, b_n \} \)  
    (i.e. output is a permutation of the input sequence)
  - \( b_1 \leq b_2 \leq \ldots \leq b_n \)
• Sorting is really only useful when it can improve the efficiency of subsequent operations…
Insertion Sorting

- Insertion-Sort(A[1..n]):
  
  for j = 2 to n
  key = A[ j ]
  i = j – 1
  while i > 0 and key ≤ A[ i ]
    A[ i + 1] = A[ i ]
    i = i – 1
  A[ i + 1 ] = key

- Does this algorithm sort A correctly?
  - Compare this with page 17 of CLRS for notation…
Correctness of Insertion Sort

**Insertion-Sort(A[1..n]):**
for j = 2 to n
  key = A[ j ]
  i = j – 1
while i > 0 and key ≤ A[ i ]
  A[ i + 1 ] = A[ i ]
  i = i – 1
  A[ i + 1 ] = key

• Use Loop Invariants
  ▪ Initialization
    • Like a “Base Case”
  ▪ Maintenance
    • Like “Inductive Step”
  ▪ Termination
    • True at end of loop

• Consider the for loop:

  • Initialization: j = 2, thus A[1 .. j-1] is in sorted order at start
  • Claim: At end of each loop, A[1 .. j] is in sorted order
    ▪ Maintenance: if A[1 .. j-1] was sorted at the start of the loop, then A[1 .. j-1] will be sorted at the end
  • Termination: At end of last loop, A[1..n] is sorted

Shabsi Walfish
NYU - Fundamental Algorithms
Summer 2005
Runtime of Insertion Sort

Insertion-Sort(A[1..n]):
for j = 2 to n
  key = A[ j ]
  i = j - 1
  while i > 0 and key ≤ A[ i ]
    A[ i + 1 ] = A[ i ]
    i = i - 1
  A[ i + 1 ] = key

- What takes time?
  - CLRS counts each op…
  - We will count uses of ≤
- Easy to see the outer loop happens n-1 times, but what about the inner one?

- “Worst case” runtime analysis: how bad could it be?
- Worst case happens if input is exactly “anti-sorted”
  - The inner loop will run from i = j-1 to 0, total of j times
  - One ≤ used per inner loop, total of $\sum_{j=2}^{n} j = \ldots$ uses
- What is the best case?
Merge Sorting 1

- Observation: It is easy to merge two pre-sorted lists
- Merge($L[1..n_1], R[1..n_2]$):
  - $n = n_1 + n_2$; $i, j = 1$
  - Create array $A[1..n]$
  - for $k = 1$ to $n$
    - if $L[i] \leq R[j]$ then  // Out of bounds = $\infty$
      - $A[k] = L[i]; i = i+1$
    - else
      - $A[k] = R[j]; j = j+1$
  - return $A$  // $A$ is now a merge of $L, R$
- Uses exactly $n = n_1 + n_2$ comparisons
Merge Sorting 2

• Intuition: “Divide and Conquer”. Chop input into smaller, easily sorted lists... then merge them
• Merge-Sort( A[ 1..n ] ):
  if n > 1 then
    p = ⌊ n/2 ⌋
    L = Merge-Sort(A[ 1 .. p ])
    R = Merge-Sort(A[ p+1 .. n ])
    return Merge(L, R)
  else return A
• Correctness follows from correctness of Merge
• How can we analyze the runtime?
Runtime of Merge Sort

Merge-Sort(A[1..n]):

if n > 1 then
  p = ⌊n/2⌋
  L = Merge-Sort(A[1..p])
  R = Merge-Sort(A[p+1..n])
  return Merge(L, R)
else return A

- Exactly n total comparison operations are performed by the call to Merge(L, R)
- How many comparisons due to the recursion?
- Write a recurrence eqn.

\[
T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n
\]
\[
T(2) = 2
\]
- To simplify, can consider only n of the form 2^i for some i
- How do we solve this?
Solving the Recurrence: Method 1

- Know the answer... then prove it using induction
  - Helps to be a psychic. Since you probably aren’t, I will tell you the answer is: \( T(n) = n \lg n \)

Proof:
1) Check basis step first: \( T(2) = 2 \lg 2 = 2 \checkmark \)
2) Assume: \( T(2^i) = 2^i \lg 2^i \) (inductive hypothesis)

Need to show: \( T(2^{i+1}) = 2^{i+1} \lg 2^{i+1} \)

By definition: \( T(2^{i+1}) = T(2^i) + T(2^i) + 2^{i+1} \)
\( = 2 \cdot (2^i \lg 2^i + 2^i) + 2^{i+1} = 2^{i+1}(1+\lg 2^i) + 2^{i+1} \)
\( = 2^{i+1} \lg 2^{i+1} + 2^{i+1} \checkmark \)

- \( T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n \), \( T(2) = 2 \)
- Consider only \( n \) of the form \( 2^i \) for some \( i \)
Solving the Recurrence: Method 2

• Recursion Trees
  ▪ See diagram in CLRS (I will draw this for you)
  ▪ Much more intuitive, but somewhat error prone
  ▪ Also easy to show that we don’t really need \( n \) of the form \( 2^i \)…

\[
T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n, \quad T(2) = 2
\]

• Consider only \( n \) of the form \( 2^i \) for some \( i \)
Solving the Recurrence: Method 3

• Algebraic Techniques (more on these in the next class)
  ▪ Yield exact solutions
  ▪ Less error prone
  ▪ Much harder for most people

• In general, main techniques are
  ▪ Telescoping
  ▪ Domain Transformations
  ▪ Range Transformations

• Can often “cheat”, and apply the “Master Theorem”

\[ T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n \quad , \quad T(2) = 2 \]

• Consider only \( n \) of the form \( 2^i \) for some \( i \)
Asymptotic Behavior

- Theoretically, constant factors don’t matter much...
  - e.g. what is faster, $4n^2 + 10$ or $n^3$ operations?
  - In practice, they often do matter though
- Primarily, we will consider the design of “scalable” algorithms that must be efficient for large inputs
  - Bio-informatics, Google, etc.
- Thus, our primary concern is the behavior of algorithms as the input size tends towards $\infty$
  - This means we should consider the asymptotic behavior of efficiency measures such as runtime
O–Notation

• Asymptotic Upper Bound
  ▪ Definition: \( f(n) = O(g(n)) \) iff there exist positive constants \( c \) and \( n_0 \) such that:

  \[ 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \]

  ▪ Intuitively, this states that some constant multiple of \( g(n) \) eventually grows faster than \( f(n) \) as \( n \) gets larger

  ▪ Be careful, the “=“ operator here is *not* equality!

• Observe that \( c \) can be arbitrary, so any constant factors in \( g(n) \) are irrelevant. Just omit them.

• Example: \( 2n + \log n = O(n) \)
Ω–Notation

• Asymptotic Lower Bound
  ▪ Definition: \( f(n) = \Omega(g(n)) \) iff there exist positive constants \( c \) and \( n_0 \) such that:

\[
0 \leq c \ g(n) \leq f(n) \text{ for all } n \geq n_0
\]

  ▪ Intuitively, this states that \( f(n) \) eventually grows faster than some constant multiple of \( g(n) \) as \( n \) gets larger
  ▪ Again, the “\(=\)” operator here is not equality!

• Observe that \( c \) can be arbitrary, so any constant factors in \( g(n) \) are irrelevant. Just omit them.

• Example: \( 2n + \log n = \Omega(n) \)
Θ—Notation

• Asymptotically Tight Bound
  ▪ Definition: \( f(n) = \Theta(g(n)) \) iff there exist positive constants \( c_1, c_2, \) and \( n_0 \) such that:

  \[
  0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0
  \]
  ▪ Intuitively, this states that \( f(n) \) eventually grows like a constant multiple of \( g(n) \) as \( n \) gets larger
  ▪ Again, the “=“ operator here is not equality!

• Observe that \( c \) can be arbitrary, so any constant factors in \( g(n) \) are irrelevant. Just omit them.

• Example: \( 2n + \log n = \Theta(n) \)
o–Notation

• **Strict Asymptotic Upper Bound**
  - Definition: \( f(n) = o(g(n)) \) iff for any positive constant \( c \) there exists a positive constant \( n_0 \) such that:

  \[
  0 \leq f(n) \leq c \; g(n) \text{ for all } n \geq n_0
  \]

  - Intuitively, this states that *any* constant multiple of \( g(n) \) eventually grows faster than \( f(n) \) as \( n \) gets larger.

  - Again, the “=“ operator here is *not* equality!

• Observe that \( c \) can be arbitrary, so any constant factors in \( g(n) \) are irrelevant. Just omit them.

• Example: \( 2n + \log n = o(n^2) \)
\( \omega \)-Notation

- Asymptotic Lower Bound
  - Definition: \( f(n) = \omega(g(n)) \) iff for any positive constant \( c \) there exists a positive constant \( n_0 \) such that:

  \[
  0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0
  \]
  - Intuitively, this states that \( f(n) \) eventually grows faster than any constant multiple of \( g(n) \) as \( n \) gets larger.
  - Again, the “\( \ll \)" operator here is not equality!

- Observe that \( c \) can be arbitrary, so any constant factors in \( g(n) \) are irrelevant. Just omit them.

- Example: \( 2n + \log n = \omega(\log n) \)
Useful Relationships

• Transitivity: \( f(n) = O(g(n)) \) and \( g(n) = O(h(n)) \) implies that \( f(n) = O(h(n)) \) (similarly...)
• Reflexivity: \( f(n) = O(f(n)) \) (similarly...)
• \( f(n) = \Theta(g(n)) \) iff \( g(n) = \Theta(f(n)) \)
• \( f(n) = O(g(n)) \) iff \( g(n) = \Omega(f(n)) \)
• \( f(n) = o(g(n)) \) iff \( g(n) = \omega(f(n)) \)
• \( f(n) = \Theta(g(n)) \) iff \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \)