CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Designing a *Reduction* Algorithm: A Simple Example

- Sometimes, we can design an algo to *reduce* a problem *A* to a problem *B*
  - Such an algorithm is called a *reduction* from *A* to *B*
  - A reduction from *A* to *B* is a kind of algo that solves *A* by using a subroutine that solves *B*—thus reducing *A* to *B*
- Here's a *very* simple example. Consider problems *A* and *B* with these specifications:

A:
- **Input: List L = [c_1 … c_n] of numbers.**
- **Output: *True* if the first element of L is 375; False otherwise.**

B:
- **Input: List M = [d_1 … d_k] of numbers.**
- **Output: *True* if the last element of M is 375; *False* otherwise.**

- What's an algorithm that would solve *A*, using a solution for *B* as a subroutine?

Designing a *Reduction* Algorithm: A Simple Example

- Sometimes, we can design an algo to *reduce* a problem *A* to a problem *B*
  - Such an algorithm is called a *reduction* from *A* to *B*
  - A reduction from *A* to *B* is a kind of algo that solves *A* by using a subroutine that solves *B*—thus reducing *A* to *B*
- Here's a *very* simple example. Consider problems *A* and *B* with these specifications:

A:
- **Input: List L = [c_1 … c_n] of numbers.**
- **Output: *True* if the first element of L is 375; False otherwise.**

B:
- **Input: List M = [d_1 … d_k] of numbers.**
- **Output: *True* if the last element of M is 375; *False* otherwise.**

- What's an algorithm that would solve *A*, using a solution for *B* as a subroutine?

Here's one possible reduction from *A* to *B*:
- **From input L to *A*, create list M = [c_1]**
- **Run a subroutine for *B* on input M. If it returns True, your algorithm for *A* should return True; if it returns False, your algorithm for *A* should return False.**

Can you analyze the complexity and explain the correctness of this reduction algorithm?

Designing a *Reduction* Algorithm: A Simple Example

- Sometimes, we can design an algo to *reduce* a problem *A* to a problem *B*
  - Such an algorithm is called a *reduction* from *A* to *B*
  - A reduction from *A* to *B* is a kind of algo that solves *A* by using a subroutine that solves *B*—thus reducing *A* to *B*
- Here's a *very* simple example. Consider problems *A* and *B* with these specifications:

A:
- **Input: List L = [c_1 … c_n] of numbers.**
- **Output: *True* if the first element of L is 375; False otherwise.**

B:
- **Input: List M = [d_1 … d_k] of numbers.**
- **Output: *True* if the last element of M is 375; *False* otherwise.**

You'll be designing a reduction for Project 2! Some Important notes:
- **This example may be much simpler than the reduction you design!**
- **Your write-up of your reduction will need to be more detailed than the notes on the previous slide are.**
- **As always, please feel free to ask me questions!**

Business

- Grading update:
  - SA2, SA3 in progress
- PS2 out, extended deadline: due Oct. 26
- Project 2 Lookahead out
  - Plenty to get started on!
  - First part due Oct. 24
  - Please read over instructions and let me know if there are any questions!
- The full Proj2 assignment document that I post will be lengthy
  - It is meant to be a *teaching* and *reference* document in places, with many hints and specific examples
  - I hope you find it useful!
- Please email me with Proj2 teams by end of day today!

Business: Project 2

- Project 2 out *very soon*
  - Multi-stage project, with final due date in early November
- **Project 2 is to be done in teams of 4**
  - If you'd like my help finding a team for you, please let me know!
- Parts of Project 2:
  1. **Design Exhaustive Search Algorithms**: Your team will collectively design exhaustive search algorithms for 8 problems.
  2. **Improve Time Efficiency**: Your team will pick one of the problems and make your exhaustive search algorithm more efficient.
  3. ***Reduction***: For the same problem chosen for part 2 above, you will *reduce* that problem to one of the other seven problems from part 1.
  4. **Create and Give a Presentation**: Your team will present work from the previous three parts of the assignment, *using loop invariants* where appropriate to explain correctness.

Hint: Your team may want to be strategic about which of the 8 problems you choose to focus on for your improvements, reduction, and presentation. Pick a problem for which you can do good work!

The green-shaded ones are examples of *polynomial time* classes—upper bounded by n^k for some constant *k*. Problems solvable in polynomial time are considered *tractable.* (More about this later in the semester!)

Time Complexity Classes Illustrated!

Complexity Class	*What we call it*	*Example algorithms / objects*
O(1)	Constant	Print "Hello, World!"; stack operations [and much, much more—be careful!]
O(lg n)	Log time	Binary search
O(n)	Linear	Exhaustive search of an array (linear search); Merge (as used in Mergesort)
O(n lg n)	n lg n	Mergesort; Heapsort [Recall: sorting can be done in θ(n lg n)]
O(n^2)	n-squared; quadratic	Insertion / selection / bubble sort; several graph algos
O(n^3)	n-cubed; cubic	*My favorite algorithm*! (a graph algo)
O(2^n)	Exponential	Number of *subsets* of a set of size n
O(n!)	Factorial	Number of orderings / permutations of elements of a list of length n

O(lg n): Binary Search

- *Binary search*: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
  - How could search be made more efficient on a sorted array?

Problem:

Input: sorted array *A*,
value *v* for which to search

Output: index *i* such that *v* = *A*[*i*]
or the special value NIL if
v* does not appear in *A

O(lg n): Binary Search

- *Binary search*: *Divide-and-conquer* search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
  - How could search be made more efficient on a sorted array?

A quick vocabulary note:

Divide-and-conquer **refers to algorithms that break problems down into subproblems of the same type**

We'll go into this more when we look at recursion—recursive algos are often divide-and-conquer!

O(lg n): Binary Search

- *Binary search*: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
  - How could search be made more efficient on a sorted array?

```
Algorithm: BinSrch(A[0..n-1],v)
  L = 0; R = n-1
  while L ≤ R do
    mid = (L+R)/2 # int division
    if v == A[mid]
      return mid
    elif v > A[mid]
      L = mid + 1
    else # must be v < A[mid]
      R = mid - 1
  return NIL
```

Problem:

Input: sorted array *A*, value *v* for which to search

Output: index *i* such that *v* = *A*[*i*] or the special value NIL if *v* does not appear in *A*

O(lg n): Binary Search

- *Binary search*: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
  - How could search be made more efficient on a sorted array?

```
Algorithm: BinSrch(A[0..n-1],v)
  L = 0; R = n-1
  while L ≤ R do
    mid = (L+R)/2 # int division
    if v == A[mid]
      return mid
    elif v > A[mid]
      L = mid + 1
    else # must be v < A[mid]
      R = mid - 1
  return NIL
```

What would a complexity argument be for this algorithm?

O(lg n): Binary Search

- *Binary search*: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
  - How could search be made more efficient on a sorted array?

```
Algorithm: BinSrch(A[0..n-1],v)
  L = 0; R = n-1
  while L ≤ R do
    mid = (L+R)/2 # int division
    if v == A[mid]
      return mid
    elif v > A[mid]
      L = mid + 1
    else # must be v < A[mid]
      R = mid - 1
  return NIL
```

What would a complexity argument be for this algorithm?

- How many iterations through the while loop?
- How much work done each iteration?

Worst case time complexity: O(lg n)

O(lg n): Binary Search

- *Binary search*: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
  - How could search be made more efficient on a sorted array?

```
Algorithm: BinSrch(A[0..n-1],v)
  L = 0; R = n-1
  while L ≤ R do
    mid = (L+R)/2 # int division
    if v == A[mid]
      return mid
    elif v > A[mid]
      L = mid + 1
    else # must be v < A[mid]
      R = mid - 1
  return NIL
```

What could a ***loop invariant*** be, for a correctness argument for algorithm?

Correctness: Binary Search

- *Binary search*: How would we explain its correctness?

(Recall the text in this text box from a previous slide…)

To (informally) use loop invariants to help explain algo correctness:

- **Explain how the invariant is true before the first iteration of the loop**
- **Explain how the invariant is true after each following iteration**
- **Explain how the invariant property shows that the algo meets its specifications**

```
Algorithm: BinSrch(A[0..n-1],v)
  L = 0; R = n-1
  while L ≤ R do
    mid = (L+R)/2 # int division
    if v == A[mid]
      return mid
    elif v > A[mid]
      L = mid + 1
    else # must be v < A[mid]
      R = mid - 1
  return NIL
```

What could a ***loop invariant*** be, for a correctness argument for algorithm?

One possibility:

- **A is unchanged from original input**
- **v may occur in A[L..R], but not elsewhere in A**

<u>Recall problem specifications</u>:

- Input: sorted array *A*, value *v*
- Output: index *i* such that *v = A[i]*, or the NIL if *v* does not appear in *A*

O(lg n): Binary Search

- *Binary search*: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
- Recursion warmup! What's a recursive algo for binary search?

```
Algorithm: BinSrch(A[0..n-1],v)
 L = 0; R = n-1
 while L ≤ R do
  mid = (L+R)/2 # int division
  if v == A[mid]
   return mid
  elif v > A[mid]
   L = mid + 1
  else # must be v < A[mid]
   R = mid - 1
 return NIL
```

O(lg n): Binary Search

- *Binary search*: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
- Recursion warmup! What's a recursive algo for binary search?

```
Algorithm: BinSrch(A,v,low,high)
 if low > high
  return False
 else
  mid = (low+high)/2 # int division
  if v == A[mid]
   return True
  elif v > A[mid]
   return BinSrch(A,v,mid+1,high)
  else # must be v < A[mid]
   return BinSrch(A,v,low,mid-1)
```

Problem (*modified!*):

Input: sorted array *A*,
value *v* for which to search,
integers *low* and *high* to specify range of *A* in which to search

Output: True if *v* is an element of *A[low.. high]*,
False otherwise

O(lg n): Binary Search

- *Binary search*: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
- Recursion warmup! What's a recursive algo for binary search?

```
Algorithm: BinSrch(A,v,low,high)
 if low > high
  return False
 else
  mid = (low+high)/2 # int division
  if v == A[mid]
   return True
  elif v > A[mid]
   return BinSrch(A,v,mid+1,high)
  else # must be v < A[mid]
   return BinSrch(A,v,low,mid-1)
```

What would the initial call to this function be, to find *v* in all of *A*?

You may have noticed the specification for this is different from the original spec'n for the search problem!

We could use a *wrapper* function to make this work with the original specification

O(lg n): Binary Search

- *Binary search*: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
- Recursion warmup! What's a recursive algo for binary search?

```
Algorithm: BinSrch(A,v,low,high)
 if low > high
  return False
 else
  mid = (low+high)/2 # int division
  if v == A[mid]
   return True
  elif v > A[mid]
   return BinSrch(A,v,mid+1,high)
  else # must be v < A[mid]
   return BinSrch(A,v,low,mid-1)
```

What would the initial call to this function be, to find *v* in all of *A*?

Note: It's the same sequence A each time. Copying or altering A (with, e.g., list slicing) would take extra time

Important: The recursive cases bring the sub-problems closer to the base case where low > high

O(lg n): Binary Search

- *Binary search*: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
- Recursion warmup! What's a recursive algo for binary search?

```
Algorithm: BinSrch(A[0..n-1],v)
 L = 0; R = n-1
 while L ≤ R do
  mid = (L+R)/2 # int division
  if v == A[mid]
   return mid
  elif v > A[mid]
   L = mid + 1
  else # must be v < A[mid]
   R = mid - 1
 return NIL
```

Is the complexity of this recursive version different from the complexity of the iterative version?

O(lg n): Binary Search

- *Binary search*: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Complexity analysis: In the worst case, for input A of size n, there are lg(n) recursive calls and O(1) work each time

```
Algorithm: BinSrch(A[0..n-1],v)
 L = 0; R = n-1
 while L ≤ R do
  mid = (L+R)/2 # int division
  if v == A[mid]
   return mid
  elif v > A[mid]
   L = mid + 1
  else # must be v < A[mid]
   R = mid - 1
 return NIL
```

Worst case time complexity: θ(lg n)

Are the ideas about complexity on this slide new to you?

We'll talk *much* more about them as the semester goes along!

This isn't Mergesort! It's just the Merge algo used in Mergesort

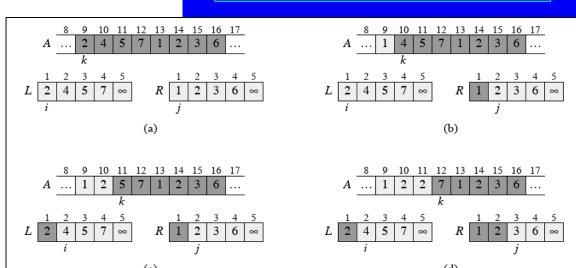
See CLRS, pg. 31

O(n): Merge

- See CLRS page 34 for specfications / loop invariant
  - In the function call *Merge(A,p,q,r)*, what are *A, p, q,* and *r*?

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

To me, one cool thing about this algo is how effective it can be to do a constant amount of work for each element in the input!
(Sorta like with stack operations—a powerful data structure with fast operations!)



This isn't Mergesort! It's just the Merge algo used in Mergesort

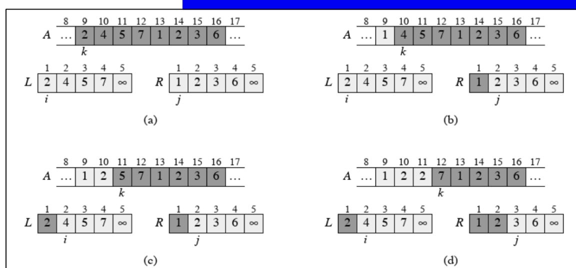
See CLRS, pg. 31

O(n): Merge

- See CLRS page 34 for specfications / loop invariant
  - In the function call *Merge(A,p,q,r)*, what are *A, p, q,* and *r*?

```
MERGE(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
  for i = 1 to n1
      L[i] = A[p + i - 1]
  for j = 1 to n2
      R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

- How would you explain the time complexity of this algorithm?
- What is the algorithm's space complexity?



See CLRS, pg. 34

O(n lg n): Mergesort

- Mergesort is a *classic* example of an *n lg n* algorithm
  - Algo idea: Repeatedly split input list in half, sort each half separately, then Merge the two halves together
  - Uses the *Merge* function as a subroutine
  - Pretty fast algo, because *Merge* is O(n)
- Pseudocode, from CLRS:

```
MERGE-SORT(A, p, r)
  if p < r
      q = ⌊(p + r)/2⌋
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q + 1, r)
      MERGE(A, p, q, r)
```