Divide and Conquer Algorithms
Divide-and-Conquer

The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions
Divide-and-Conquer Technique (cont.)

- Divide
- Conquer Technique

- a problem of size $n$
  (instance)

- subproblem 1
  of size $n/2$

- a solution to
  subproblem 1

- subproblem 2
  of size $n/2$

- a solution to
  subproblem 2

- a solution to
  the original problem

It generally leads to a recursive algorithm!
Linear Search

**Problem:** Given a list of N values, determine whether a given value X occurs in the list.

For example, consider the problem of determining whether the value 55 occurs in:

```
  1  2  3  4  5  6  7  8
  17 31  9  73  55 12 19  7
```

There is an obvious, correct algorithm:

start at one end of the list,
if the current element doesn't equal the search target, move to the next element,
stopping when a match is found or the opposite end of the list is reached.

Basic principle: divide the list into the current element and everything before (or after) it; if current isn't a match, search the other case
Linear Search

algorithm LinearSearch takes number X, list number L, number Sz

# Determines whether the value X occurs within the list L.
# Pre: L must be initialized to hold exactly Sz values
#

    # Walk from the upper end of the list toward the lower end,
    # looking for a match:

while Sz > 0 AND L[Sz] != X
    Sz := Sz - 1
endwhile

if Sz > 0    # See if we walked off the front of the list
    display true  # if not, got a match
else
    display false  # if so, no match

halt
Binary Search

Repeatedly halving the size of the “search space” is the main idea behind the method of binary search.

An item in a sorted array of length $n$ can be located with just $\log_2 n$ comparisons.
Binary Search

Repeatedly halving the size of the “search space” is the main idea behind the method of binary search.

An item in a sorted array of length $n$ can be located with just $\log_2 n$ comparisons.

“Savings” is significant!

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>10000</td>
<td>13</td>
</tr>
</tbody>
</table>
Binary search: target $x = 70$

$v = [12, 15, 33, 35, 42, 45, 51, 62, 73, 75, 86, 98]$

$L: \ 1$

$Mid: \ 6$

$R: \ 12$

$v(Mid) \leq x$

So throw away the left half...
Binary search: target $x = 70$

$v$

```
1   2   3   4   5   6   7   8   9   10  11  12
12  15  33  35  42  45  51  62  73  75  86  98
```

$x < v(Mid)$

So throw away the right half...
Binary search: target \( x = 70 \)

So throw away the left half…
Binary search: target $x = 70$

So throw away the left half...
Binary search: target $x = 70$

$L$: 8

Mid: 8

$R$: 9

Done because $R - L = 1$
function L = BinarySearch(a,x)
# x is a row n-vector with x(1) < ... < x(n)
# where x(1) <= a <= x(n)
# L is the index such that x(L) <= a <= x(L+1)

L = 1;  R = length(x);
# x(L) <= a <= x(R)
while R-L > 1
    mid = floor((L+R)/2);
    # Note that mid does not equal L or R.
    if a < x(mid)
        # x(L) <= a <= x(mid)
        R = mid;
    else
        # x(mid) <= a <= x(R)
        L = mid;
    end
end
Binary search is efficient, but how do we sort a vector in the first place so that we can use binary search?

- Many different algorithms out there...
- Let’s look at **merge sort**
- An example of the “divide and conquer” approach
If I have two helpers, I’d...

- Give each helper half the array to sort
- Then I get back the sorted subarrays and merge them.

What if those two helpers each had two sub-helpers?

And the sub-helpers each had two sub-sub-helpers? And...
Subdivide the sorting task

HEMGGBKAKQFLPLPDRCJN

HEMGGBKAKQ FLPLPDRCJN
Subdivide again
And again
And one last time
Now merge
And merge again
And again
And one last time
Done!
function y = mergeSort(x)
# x is a vector. y is a vector
# consisting of the values in x
# sorted from smallest to largest.

n = length(x);
if n==1
    y = x;
else
    m  = floor(n/2);
yL = mergeSortL(x(1:m));
yR = mergeSortR(x(m+1:n));
y  = merge(yL,yR);
end
The central sub-problem is the **merging** of two sorted arrays into one single sorted array.
Merge

x: 12 33 35 45

y: 15 42 55 65 75

z: [empty]

ix: 1
iy: 1
iz: 1

ix<=4 and iy<=5: x(ix) <= y(iy) ??
Merge

ix\leq4 \text{ and } iy\leq5: x(ix) \leq y(iy) \text{ YES}
Merge

x:  
  12  33  35  45

y:  
  15  42  55  65  75

z:  
  12

ix:  
  2

iy:  
  1

iz:  
  2

ix <= 4 and iy <= 5:  x(ix) <= y(iy)  ???
Merge

ix<=4 and iy<=5: x(ix) <= y(iy)  NO
ix <= 4 and iy <= 5: x(ix) <= y(iy) ???
ix <= 4 and iy <= 5: x(ix) <= y(iy)  YES
Merge

\[ \text{ix} \leq 4 \quad \text{and} \quad \text{iy} \leq 5: \quad x(\text{ix}) \leq y(\text{iy}) \quad ??? \]
Merge

\[\text{ix} \leq 4 \text{ and } \text{iy} \leq 5: \ x(\text{ix}) \leq y(\text{iy}) \quad \text{YES}\]
ix<=4 and iy<=5: x(ix) <= y(iy) ???
Merge

ix <= 4 and iy <= 5: x(ix) <= y(iy) NO
ix <= 4 and iy <= 5: x[ix] <= y[iy] ??
ix <= 4 and iy <= 5: x(ix) <= y(iy)  YES
Merge

\[
\begin{align*}
\text{x:} & \quad 12 \quad 33 \quad 35 \quad 45 \\
\text{y:} & \quad 15 \quad 42 \quad 55 \quad 65 \quad 75 \\
\text{z:} & \quad 12 \quad 15 \quad 33 \quad 35 \quad 42 \quad 45 \quad \ldots \\
\end{align*}
\]

\[
i_x > 4
\]
Merge

\[ \text{ix} > 4: \text{take } y(\text{iy}) \]
Merge

x: 12 33 35 45

y: 15 42 55 65 75

z: 12 15 33 35 42 45 55

iy <= 5

ix: 5

iy: 4

iz: 8
Merge

\[ iy \leq 5 \]
Merge

x: 

12 33 35 45

ix: 5

y: 

15 42 55 65 75

iy: 5

z: 

12 15 33 35 42 45 55 65

iz: 9

iy <= 5
Merge

x: 12 33 35 45

y: 15 42 55 65 75

z: 12 15 33 35 42 45 55 65 75

iy <= 5

ix: 5

iy: 5

iz: 9
function z = merge(x,y)
nx = length(x); ny = length(y);
z = zeros(1,nx+ny);
ix = 1; iy = 1; iz = 1;
function z = merge(x,y)
  nx = length(x); ny = length(y);
  z = zeros(1, nx+ny);
  ix = 1; iy = 1; iz = 1;
  while ix<=nx && iy<=ny

end
# Deal with remaining values in x or y
function z = merge(x,y)
  nx = length(x); ny = length(y);
  z = zeros(1, nx+ny);
  ix = 1; iy = 1; iz = 1;
  while ix<=nx && iy<=ny
    if x(ix) <= y(iy)
      z(iz)= x(ix); ix=ix+1; iz=iz+1;
    else
      z(iz)= y(iy); iy=iy+1; iz=iz+1;
    end
  end
  # Deal with remaining values in x or y
function z = merge(x,y)

nx = length(x); ny = length(y);

z = zeros(1, nx+ny);
ix = 1; iy = 1; iz = 1;
while ix<=nx && iy<=ny
    if x(ix) <= y(iy)
        z(iz) = x(ix); ix=ix+1; iz=iz+1;
    else
        z(iz) = y(iy); iy=iy+1; iz=iz+1;
    end
end

while ix<=nx    # copy remaining x-values
    z(iz) = x(ix); ix=ix+1; iz=iz+1;
end

while iy<=ny    # copy remaining y-values
    z(iz) = y(iy); iy=iy+1; iz=iz+1;
end
function y = mergeSort(x)
# x is a vector. y is a vector
# consisting of the values in x
# sorted from smallest to largest.

n = length(x);
if n==1
    y = x;
else
    m  = floor(n/2);
yL = mergeSortL(x(1:m));
yR = mergeSortR(x(m+1:n));
y  = merge(yL,yR);
end
function y = mergeSortL(x)
# x is a vector. y is a vector consisting of the values in x sorted from smallest to largest.

n = length(x);
if n==1
    y = x;
else
    m  = floor(n/2);
    yL = mergeSortL_L(x(1:m));
    yR = mergeSortL_R(x(m+1:n));
    y  = merge(yL,yR);
end
function y = mergeSortL_L(x)
# x is a vector. y is a vector
# consisting of the values in x
# sorted from smallest to largest.

n = length(x);
if n==1
    y = x;
else
    m = floor(n/2);
yL = mergeSortL_L_L(x(1:m));
yR = mergeSortL_L_R(x(m+1:n));
y = merge(yL,yR);
end

There should be just one mergeSort function!
function y = mergeSort(x)
# x is a vector. y is a vector
# consisting of the values in x
# sorted from smallest to largest.

n = length(x);
if n==1
    y = x;
else
    m  = floor(n/2);
    yL = mergeSort(x(1:m));
    yR = mergeSort(x(m+1:n));
    y  = merge(yL,yR);
end
function y=mergeSort(x)
n=length(x);
if n==1
    y=x;
else
    m=floor(n/2);
yL=mergeSort(x(1:m));
yR=mergeSort(x(m+1:n));
y=merge(yL,yR);
end
Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search
- Multiplication of large integers
- Matrix multiplication: Strassen’s algorithm
- Closest-pair algorithm