Linear Search: HOW

Starting at the beginning of the list, compare items one by one with \( x \) until find it or reach the end.

**procedure** linear search (\( x: \text{integer}, \ a_1, \ a_2, \ ..., \ a_n: \text{distinct integers} \) )

\[
i := 1
\]

while (\( i \leq n \) and \( x \neq a_i \))

\[
i := i+1
\]

if \( i \leq n \) then location := i

else location := 0

return location

{ location is the subscript of the term that equals \( x \), or is 0 if \( x \) is not found }
The time it takes to find \( x \) (or determine it is not present) depends on the number of probes, that is the number of list entries we have to retrieve and compare to \( x \).

### How many probes do we make when doing Linear Search on a list of size \( n \):

- if \( x \) happens to equal the **first** element in the list?
- if \( x \) happens to equal the **last** element in the list?
- if \( x \) happens to equal an element somewhere in the **middle** of the list?
- if \( x \) **doesn't equal any** element in the list?
How fast is Linear Search: WHEN

- **Best case:** 1 probe  
  - target appears first

- **Worst case:** n probes  
  - target appears last or not at all

- **Average case:** n/2 probes  
  - target appears in the middle,  
    (expect to have to search about half of the array ... more on expected value later in the course)

Running time depends on more than size of the input!

Rosen p. 220
procedure binary search (x: integer, a₁, a₂, ..., aₙ: increasing integers )
i := 1
j := n
while i<j
    m := floor( (i+j)/2 )
    if x > aₘ then i := m+1
    else j := m
if x=aᵢ then location := i
else location := 0
return location

{ location is the subscript of the term that equals x, or is 0 if x is not found }
Number of comparisons (probes) depends on number of iterations of loop, hence size of set.

<table>
<thead>
<tr>
<th>After … iterations of loop</th>
<th>(Max) size of list &quot;in play&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>1</td>
<td>n/2</td>
</tr>
<tr>
<td>2</td>
<td>n/4</td>
</tr>
<tr>
<td>3</td>
<td>n/8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>??</td>
<td>1</td>
</tr>
</tbody>
</table>
How fast is Binary Search: WHEN

Rosen page 220, example 3

Number of comparisons (probes) depends on number of iterations of loop, hence size of set.

<table>
<thead>
<tr>
<th>After ... iterations of loop</th>
<th>(Max) size of list &quot;in play&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>1</td>
<td>n/2</td>
</tr>
<tr>
<td>2</td>
<td>n/4</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>ceil(log₂ n)</td>
<td></td>
</tr>
</tbody>
</table>

Rewrite this formula in order notation:

A. Θ(log n)
B. Θ(log n + 1)
C. Θ(n)
D. Θ(log₁₀ n)
E. None of the above
Comparing linear search and binary search

Rosen pages 220-221

<table>
<thead>
<tr>
<th></th>
<th>Linear Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assumptions</td>
<td>None</td>
<td>Sorted list</td>
</tr>
<tr>
<td># probes in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* best case</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$ or $\Theta(\log n)$</td>
</tr>
<tr>
<td>* worst case</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>* average case</td>
<td>$\Theta(n)$</td>
<td>??</td>
</tr>
</tbody>
</table>

Best case analysis depends on whether we check if midpoint agrees with target right away or wait until list size gets to 1
## Comparing linear search and binary search

<table>
<thead>
<tr>
<th></th>
<th>Linear Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assumptions</td>
<td>None</td>
<td>Sorted list</td>
</tr>
<tr>
<td># probes in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* best case</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$or $\Theta(\log n)$</td>
</tr>
<tr>
<td>* worst case</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>* average case</td>
<td>$\Theta(n)$</td>
<td>??</td>
</tr>
</tbody>
</table>
Comparing linear search and binary search

<table>
<thead>
<tr>
<th></th>
<th>Linear Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assumptions</td>
<td>None</td>
<td>Sorted list</td>
</tr>
<tr>
<td># probes in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* best case</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$ or $\Theta(\log n)$</td>
</tr>
<tr>
<td>* worst case</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>* average case</td>
<td>$\Theta(n)$</td>
<td>??</td>
</tr>
</tbody>
</table>

Is it worth it to sort our list first?
Computing the big-O class of algorithms

How to deal with …

Basic operations

Consecutive (non-nested) code

Loops (simple and nested)

Subroutines
Computing the big-O class of algorithms

How to deal with …

**Basic operations**: operation whose time doesn't depend on input

**Consecutive (non-nested) code**: one operation followed by another

**Loops (simple and nested)**: while loops, for loops

**Subroutines**: method calls
Consecutive (non-nested) code: Run Prog₁ followed by Prog₂

If Prog₁ takes \( O(f(n)) \) time and Prog₂ takes \( O(g(n)) \) time, what's the big-O class of runtime for running them consecutively?

A. \( O(f(n) + g(n)) \) [[sum]]
B. \( O(f(n)g(n)) \) [[multiplication]]
C. \( O(g(f(n))) \) [[function composition]]
D. \( O(\max(f(n), g(n))) \)
E. None of the above.
Computing the big-O class of algorithms

Simple loops: \[ \text{while (Guard Condition)} \]
\[ \text{Body of the Loop} \]

What's the runtime?

A. Constant
B. Same order as the number of iterations through the loop.
C. Same order as the runtime of the guard condition
D. Same order as the runtime of the body of the loop.
E. None of the above.
Computing the big-O class of algorithms

Simple loops:

```
while (Guard Condition)
    Body of the Loop
```

If Guard Condition uses basic operations and body of the loop is constant time, then runtime is of the same order as the number of iterations.
Computing the big-O class of algorithms

Nested code:

```
while (Guard Condition)
    Body of the Loop,
    May contain other loops, etc.
```

If Guard Condition uses basic operations and body of the loop has runtime $O(T_2)$ in the worst case, then runtime is

$$O(T_1 T_2)$$

where $T_1$ is the bound on the number of iterations through the loop.
Subroutine Call method S on (some part of) the input.

If sub-routine S has runtime $T_S(n)$ and we call S at most $T_1$ times,

A. Total time for all uses of S is $T_1 + T_S(n)$
B. Total time for all uses of S is $\max(T_1, T_S(n))$
C. Total time for all uses of S is $T_1 T_S(n)$
D. None of the above
Subroutine Call method S on (some part of) the input.

If sub-routine S has runtime $O(T_S(n))$ and if we call S at most $T_1$ times, then runtime is

$$O(T_1 T_S(m))$$

where $m$ is the size of biggest input given to S.

Distinguish between the size of input to subroutine, $m$, and the size of the original input, $n$, to main procedure!
Selection Sort (MinSort) Pseudocode

Before, we counted comparisons, and then went to big-O

```
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( a_j < a_m ) then m := j
    interchange a_i and a_m

{ a_1, ..., a_n is in increasing order}
```

(n-1) + (n-2) + ... + (1)  
= n(n-1)/2  
∈ O(n^2)
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( a_j < a_m ) then m := j
    interchange a_i and a_m
{ a_1, ..., a_n is in increasing order}

Now, straight to big O

Strategy: work from the inside out
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >= 2)
for i := 1 to n-1
    m := i
    for j := i+1 to n
        if (a_j < a_m) then m := j
    interchange a_i and a_m

{ a_1, ..., a_n is in increasing order}

Strategy: work from the inside out
procedure selection sort(a₁, a₂, ..., aₙ: real numbers with n >= 2)
for i := 1 to n-1
    m := i
    for j := i+1 to n
        if (aⱼ < aₘ) then m := j
    interchange aᵢ and aₘ

{ a₁, ..., aₙ is in increasing order}

Now, straight to big O

Strategy: work from the inside out

Simple for loop, repeats n-i times
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >=2 )
for i := 1 to n-1
  m := i
  for j:= i+1 to n
    if ( a_j < a_m ) then m := j
  interchange a_i and a_m

{ a_1, ..., a_n is in increasing order}
procedure selection sort(a₁, a₂, ..., aₙ: real numbers with n ≥ 2)
for i := 1 to n-1
  m := i
  for j := i+1 to n
    if (aⱼ < aₘ) then m := j
interchange aᵢ and aₘ
{ a₁, ..., aₙ is in increasing order}

Strategy: work from the inside out

Worst case: when i = 1, O(n)
procedure selection sort(a₁, a₂, ..., aₙ: real numbers with n >=2 )
for i := 1 to n-1
  m := i
  for j:= i+1 to n
    if (aᵢ < aⱼ) then
      m := j
  interchange aᵢ and aᵢ
does { a₁, ..., aₙ is in increasing order}

Strategy: work from the inside out
## Selection Sort (MinSort) Pseudocode

Now, straight to big O

```plaintext
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >= 2 )
for i := 1 to n-1

{ a_1, ..., a_n is in increasing order}
```

*Strategy: work from the inside out*
Selection Sort (MinSort) Pseudocode

procedure selection sort(a₁, a₂, ..., aₙ: real numbers with n >=2 )
for i := 1 to n-1
  m := i
  for j:= i+1 to n
    if (aₗ < aᵢ) then
      m := j
  interchange aᵢ and aₗ
{ a₁, ..., aₙ is in increasing order}

Now, straight to big O

Strategy: work from the inside out

O(n)

Nested for loop, repeats O(n) times

Total: O(n²)
Next Time

Analyzing algorithms that solve other problems (besides sorting and searching)

Designing better algorithms
• pre-processing
• re-use of computation