Time Analysis of Sorting and Searching Algorithms

CSE21 Winter 2017, Day 5 (B00), Day 3 (A00)

January 20, 2017

http://vlsicad.ucsd.edu/courses/cse21-w17
procedure binary search (x: integer, a_1, a_2, ..., a_n: increasing integers )
  i := 1
  j := n
  while i<j
    m := floor( (i+j)/2 )
    if x > a_m then i := m+1
    else j := m
    if x=a_i 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 }
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>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. \( \Theta(\log n) \)
B. \( \Theta(\log n + 1) \)
C. \( \Theta(n) \)
D. \( \Theta(\log_{10} n) \)
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

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>
## 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><strong>Assumptions</strong></td>
<td>None</td>
<td>Sorted list</td>
</tr>
<tr>
<td><strong># probes in</strong></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?
Determining the big-O class of algorithms

How to deal with …

Basic operations

Consecutive (non-nested) code

Loops (simple and nested)

Subroutines
Determining 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 Prog$_1$ followed by Prog$_2$

If Prog$_1$ takes $O(f(n))$ time and Prog$_2$ 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.
Determining the big-O class of algorithms

Simple loops:

```
while (Guard Condition)
    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.
Determining 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.
Determining the big-O class of algorithms

Nested code:

\begin{center}
\textbf{while (Guard Condition)}
\end{center}

\begin{center}
Body of the Loop, \\
May contain other loops, etc.
\end{center}

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

\[ O(T_1T_2) \]

where $T_1$ is the bound on the number of \textit{iterations} through the loop.

Product rule
Determining the big-O class of algorithms

Subroutine Call method S on (some part of) the input.

If subroutine 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
Determining the big-O class of algorithms

**Subroutine** Call method S on (some part of) the input.

If subroutine S has runtime \( O(S(n)) \) 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}
```

\[(n-1) + (n-2) + \ldots + (1)\]
\[= n(n-1)/2\]
\[\in O(n^2)\]
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
    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}
Selection Sort (MinSort) Pseudocode

Now, straight to big O

```
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

Simple for loop, repeats n-i times

O(1)
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
        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

O(n-i), but i ranges from 1 to n-1
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}

Worst case: when i =1, O(n)

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

Now, straight 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
    O(1)
    for j := i+1 to n
        if a_j < a_i then
            m := j
            O(n)
    O(1)
    interchange a_i and a_m
    O(1)
    { 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

O(n)

{ 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
  m := i
  for j := i+1 to n
    if (a_j < a_i) then
      m := j
  interchange a_i and a_m

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

Nested for loop, repeats O(n) times

Total: O(n^2)

Strategy: work from the inside out
Next Time

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

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

HW 2 due **tonight 11:59pm**.

Submit assignments through **Gradescope** – one submission per group.