Algorithms

Ioan Despi
despi@turing.une.edu.au

University of New England

July 14, 2013
Outline
Introduction

- algorithm, [algo-ridhm], n. a rule for solving a mathematical problem in a finite number of steps.
  [Root: Late Latin *algorismus*, from the Arabic name Al-Khowarizmi, a 9th century mathematician Abu Jafar Mohammed ben Musa.]
- A computer program is simply an implementation of an algorithm on a computer.
Examples

- How to behave on a date
  
  WHILE wondering how to get out without being rude
  
  DO tell him he’s a wonderful guy
  
  ENDWHILE

- How to get out of bed
  
  if mother is watching
  
  then scan her for weapons,
  
  (especially those prohibited by Geneva Convention)
  
  if result is affirmative
  
  then begin negotiations

- How to swap two numbers
  
  public void smartSwap(int ref a, int ref b) {
    a += b;
    b = a - b;
    a -= b;
  }
Examples

Euclid’s Algorithm

while \( m \) is greater than zero:
  If \( n \) is greater than \( m \), swap \( m \) and \( n \).
  Subtract \( n \) from \( m \).

\( n \) is the GCD

Example

Program in C

```c
int gcd(int m, int n)
    /* precondition: \( m > 0 \) and \( n > 0 \). Let \( g = \gcd(m, n) \). */
    { while( m > 0 )
        { /* invariant: \( \gcd(m, n) = g \) */
            if( n > m )
            { int t = m; m = n; n = t; } /* swap */
            /* \( m \geq n > 0 \) */
            m -= n;
        }
    return n;
```
Example

Program in Prolog

; The goal gcd(I, J, K) succeeds when the greatest common
; divisor of I and J is K.
gcd(I,0,I).
gcd(I,J,K) :- R is I mod J, gcd(J,R,K).

Example

Program in Java

public static long gcd(long a, long b){
    long factor= Math.max(a, b);
    for(long loop= factor;loop > 1;loop--){
        if(a % loop == 0 && b % loop == 0){
            return loop;
        }
    }
}
Motivation

• Efficient algorithms lead to efficient programs.
• Efficient programs sell better.
• Efficient programs make better use of hardware.
• Programmers who write efficient programs are more marketable than those who don’t!
Computational complexity theory is the study of the cost of solving interesting problems.

- Measure the amount of resources needed
  - time
  - space
- Two aspects:
  - Upper bounds: give a fast algorithm
  - Lower bounds: no algorithm is faster

Algorithm analysis is the analysis of resource usage of given algorithms.

- Exponential resource use is bad.
- It is better (best) to
  - Make resource usage a polynomial
  - Make that polynomial as small as possible
Algorithm analysis

1. The input and the output of the algorithm should be described.
2. The finite list of rules of the algorithm should be given.
3. A proof should be given that the algorithm will convert the input into the output in a finite number of steps.

Example: describe and analyse an algorithm which computes the sum of first $n$ positive integers

1. The input is the set $\{1, 2, 3, \ldots, n\}$ and the output is the sum of these $n$ integers.
2. The algorithm has one rule, the formula
   \[1 + 2 + 3 + \cdots + n = n(n + 1)/2\]
3. We gave a proof by induction to this rule.
Searching Algorithms

- General definition
  - Locate an element $x$ in a list of distinct elements $a_1, a_2, \ldots, a_n$, or
    - determine that it is not in the list.
  - Return the position $k$ at which $a_k$ matches $x$ or, otherwise
    - return a phantom position “$-1$” (or “$o$”) used to denote that $x$ is not found.
Sequential (Linear) Searching

Linear Searching

**Input:** unsorted sequence $a_1, a_2, \ldots, a_n$
- position of target value $x$

**Output:** subscript of entry equal to target value; 0 if not found

- **Initialize:** $i \leftarrow 1$
- **while** $(i \leq n$ and $x \neq a_i)$
  - $i \leftarrow i + 1$
  - if $i \leq n$ then location $\leftarrow i$
- else location $\leftarrow 0$

- Let $S(n)$ be the number of comparisons needed in the worst case to complete the linear search.
- The worst case is when the searched item is not found or it is in the last position, that is $S(n) = n$
The floor/ceiling of a real number

Given $x \in \mathbb{R}$,

- the **floor of** $x$, denoted by $\lfloor x \rfloor$, is the greatest integer not exceeding $x$.
- the **ceiling of** $x$, denoted by $\lceil x \rceil$, is the smallest integer not less than $x$.

**Examples**

- If $x \in \mathbb{Z}$, then $\lfloor x \rfloor = \lceil x \rceil = x$
- $\lfloor 3.2 \rfloor = 3$, $\lceil -4.5 \rceil = -4$, and $\lceil -4 \rceil = -4$
- $\lfloor 3.2 \rfloor = 3$, $\lceil -4.5 \rceil = 4$, $\lceil -4 \rceil = -4$
Binary Search

- Binary search is a **divide-and-conquer** (divide et impera) strategy.
- It searches through an **ordered** list \(a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n\) for a number \(a\) (called **key**) which may/may not be in the list.
- The strategy is to compare \(a\) with the item at the middlemost position and to report this position if the items are matched.
- If not, we only need to search one of the two (halved) sublists broken up at the middlemost point (remember: the list is sorted!).
- Which sublist is to be continued for the search depends on whether the given KEY \((a)\) is after or before the item at the middlemost position.
- Continue the binary search on the correct sublist until either a match is found, or the latest sublist is reduced to an empty list.

---

**Binary Search**

1. Define two counters \(i\) and \(j\) and set \(i = 1\) and \(j = n\) as initial values.
2. If \(j < i\), the algorithm terminates without finding \(a\).
   - If not, set \(k = \lfloor (i + j)/2 \rfloor\)
3. If \(a < a_k\), go to (4). If \(a > a_k\) go to (5). If \(a = a_k\) the algorithm found \(a\).
4. Set \(j = k - 1\) and repeat (2)
5. Set \(i = k + 1\) and repeat (2)
Binary Search

- If a list $a_m, a_{m+1}, \ldots, a_n$ is indexed from $m$ to $n$, then the middlemost position is at $k = \left\lfloor \frac{m+n}{2} \right\rfloor$.
- Suppose a given list $I(1), \ldots, I(n)$ is sorted in the increasing order, then the binary search algorithm can be rephrased as follows.

**Binary Search**

1. $F = 1, L = n$. /* $F =$first index, $L =$last index */
2. while $F \leq L$ do
   (i) find middlemost position $k = \left\lfloor \frac{F + L}{2} \right\rfloor$
   (ii) if KEY = $I(k)$ then /* match found */
       output $k$ and stop
       else if KEY > $I(k)$ then /* move to 2nd half of the list */
       $F = k + 1$
       else /* redefining starting position */
       $L = k - 1$
       /* move to 1st half of the list */
   else /* redefine end position */
3. output the phantom position “−1”. /* redefining end position */
Example

Find 13 from the ordered list \( \{1, 3, 5, 7, 9, 13, 15, 17, 19\} \) with both the binary search and the sequential search. Give the number of comparisons needed in both cases.

**Binary search.** 3 comparisons are needed as can be seen from the diagram below.

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I(i) )</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>13</td>
<td>15</td>
<td>17</td>
<td>19</td>
</tr>
</tbody>
</table>

middlemost: \( 5 = \left\lfloor \frac{1 + 9}{2} \right\rfloor \)

\( 9 < 13 : \)

middlemost: \( 7 = \left\lfloor \frac{6 + 9}{2} \right\rfloor \)

\( 15 > 13 : \)

\( 13 = 13: \) match found in 3 comparisons
Example

Find 13 from the ordered list \{1, 3, 5, 7, 9, 13, 15, 17, 19\} with both the binary search and the sequential search. Give the number of comparisons needed in both cases.

Sequential search. 6 comparisons are needed in this case, see below:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I(i)$</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>13</td>
<td>15</td>
<td>17</td>
<td>19</td>
</tr>
</tbody>
</table>

→

search stops after finding the match after 6 comparisons
Complexity of Binary Search

Let \( B(n) \) be the maximum number of comparisons the binary search needs to complete the search for an ordered list of \( n \) items.

Then, for \( m, n \in \mathbb{N} \) with \( m \geq 1 \),

(i) \( B(1) = 1 \)

(ii) \( B(m) \leq B(n) \) if \( m \leq n \)

(iii) \( B(2m) = 1 + B(m) \)

(iv) \( B(2m + 1) = 1 + B(m) \)

Hence we have
\[
B(2^k) = B(2^{k-1}) + 1 = B(2^{k-2}) + 2 = \ldots = B(2^0) + k = 1 + k \text{ for } k \geq 0.
\]

For any \( n \in \mathbb{N} \) with \( n \geq 1 \), \( \exists k \in \mathbb{N} \) such that \( 2^k \leq n < 2^{k+1} \).

Taking a \( \log_2 \) on both sides of the inequality we obtain \( k \leq \log_2 n < k + 1 \).

Hence
\[
k = \lfloor \log_2 n \rfloor \iff 2^k \leq n < 2^{k+1}.
\]
Complexity of Binary Search

From (ii) we thus obtain

\[ B(2^k) \leq B(n) \leq B(2^{k+1}) \]

which gives \( k + 1 \leq B(n) \leq k + 2 \), or simply

\[ \lfloor \log_2 n \rfloor + 1 \leq B(n) \leq \lfloor \log_2 n \rfloor + 2. \]

Hence for \( n \geq 4 \), we have \( \log_2 n \geq 2 \) and \( \log_2 n \leq \lfloor \log_2 n \rfloor + 1 \) for \( n \geq 1 \) so

\[ \log_2 n \leq B(n) \leq 2 \log_2 n. \]

Thus \( B(n) = \mathcal{O}(\log_2 n) \) and \( \log_2 n = \mathcal{O}(B(n)) \).

We note that in the literature of computer science, \( \log_2 n \) is often abbreviated to \( \log n \).
Complexity of Binary Search

- Our analysis shows that binary search can be done in time proportional to the log of the number of items in the list.
- This is considered very fast when compared to linear or polynomial algorithms.
- The table below compares the number of operations that need to be performed for algorithms of various time complexities.

<table>
<thead>
<tr>
<th>n</th>
<th>log n</th>
<th>$n^2$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>25</td>
<td>32</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>100</td>
<td>1024</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>400</td>
<td>1048576</td>
</tr>
<tr>
<td>50</td>
<td>6</td>
<td>2500</td>
<td>1.1E+15</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
<td>10000</td>
<td>1.3E+30</td>
</tr>
<tr>
<td>200</td>
<td>8</td>
<td>40000</td>
<td>1.6E+60</td>
</tr>
<tr>
<td>500</td>
<td>9</td>
<td>250000</td>
<td>too big</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>1E+06</td>
<td>too big</td>
</tr>
<tr>
<td>2000</td>
<td>11</td>
<td>4E+06</td>
<td>too big</td>
</tr>
<tr>
<td>5000</td>
<td>13</td>
<td>25E+06</td>
<td>too big</td>
</tr>
</tbody>
</table>

In python:

```
amth140@turing> python
Python 2.7.3 ...
>>> 2**500
3273390607896141870013189696827599152216642046
8969682759915221664204604306478948329136809613
3796404674554883270092325904157150886684127560
2590415715088668412756007100921725654588539305
3328527589376L
```


**Performance of Sorting Algorithms**

- A number of sorting algorithms will be introduced in the tutorials over the semester:
  - bubble sort, insertion sort, selection sort, merge sort, quick sort.

- In sorting a list of \( n \) items, the performance of these sorting algorithms, for simplicity, will be based purely on the number of comparisons involved during the sorting process.

- All sorting algorithms require a total of \( \mathcal{O}(n^2) \) comparisons in the corresponding worst (i.e., the most “laborious”) cases.

- Let \( M(n) \) be number of comparisons needed for the merge sort in the worst case, and \( \overline{Q}(n) \) be the average number of comparisons needed for the quick sort, then both \( M(n) \) and \( \overline{Q}(n) \) are \( \mathcal{O}(n \log_2 n) \).

- More precisely, however, one can show for sorting a list of \( n \) (\( \geq 1 \)) items

\[
M(n) \leq 4n \log_2 n, \quad \overline{Q}(n) \leq (2 \ln 2)n \log_2 n
\]