“Big O” Notation

Analysis of algorithms
identifying critical steps
techniques for counting steps
“big O” notation
Examples
Reading: Chapter 15 of NTO

Comparing Sort Algorithms

- In the first lecture (“What is Computer Science”) there was a table that showed the number of steps used in two sorting algorithms:

<table>
<thead>
<tr>
<th></th>
<th>n</th>
<th>n^2/2</th>
<th>log₂ n</th>
<th>n log₂ n</th>
</tr>
</thead>
<tbody>
<tr>
<td>poker hand</td>
<td>5</td>
<td>12</td>
<td>2.32</td>
<td>12</td>
</tr>
<tr>
<td>bridge hand</td>
<td>13</td>
<td>85</td>
<td>3.70</td>
<td>48</td>
</tr>
<tr>
<td>full deck</td>
<td>52</td>
<td>1352</td>
<td>5.70</td>
<td>296</td>
</tr>
</tbody>
</table>

- The number in the n column is the number of cards in a hand
- The numbers in red are the number of steps taken by two different algorithms
- Today we’ll see how computer scientists analyze descriptions of algorithms to derive formulas like \( n^2 \) and \( n \log_2 n \)

Mathematical Notation

- Several of the slides in this lecture will use notation from discrete math at UO: MTH 231-232-233 (taken by all CIS majors)
- Do not be alarmed if you have not seen some of these symbols before
- Our goals in CIS 170 are to understand where the equations come from what the equations mean
- You do not need to know how to derive the equations or how to solve them

Mathematical Notation (cont’d)

- A notation used in this lecture describes **summation**
- If \( A \) is an array (list) of numbers, then

\[
\sum A
\]

means “the sum of all numbers in \( A \)”

- Sometimes it’s necessary to specify which items to sum; the notation

\[
\sum_{i=j}^{k} A_i
\]

think “for statement” in Ruby or pseudo-code

means “the sum of \( A_j \) through \( A_k \)”

- The notation for the sum of all \( n \) numbers in an array \( A \) is

\[
\sum_{i=0}^{n-1} A_i
\]
Matching Pairs of Integers

- The example we’ll use to illustrate the process of analyzing an algorithm comes from NTO
- The algorithm scans a list of integers to see if any two numbers are the same

```plaintext
for i = 0 to n-2
  for j = i+1 to n-1
    if A[i] = A[j]
      print i and j
      exit
    continue
```

- The pseudo-code above is slightly different than the algorithm in NTO
  - arrays in most programming languages (including Ruby) start with A₀, not A₁
  - our outer loop counts from 0 to n-2 instead of 1 to n-1

Matching Pairs in Ruby

- Here’s how the loops can be written in Ruby:

```ruby
for i in 0..a.length-2
  for j in i+1..a.length-1
    if a[i] == a[j]
      puts "match: a[#{i}] = #{a[i]}, a[#{j}] = #{a[j]}
      return
    end
  end
end
```

- Notes:
  - Ruby variable names always start with lower case letters (a instead of A)
  - i..j is an “iterator”
  - puts means “put string”
  - Important! The comparison is a[i] == a[j] and not a[i] = a[j]

Loops in Matching Pairs

- The loop structure of matching pairs is very similar to the nested loops of the insertion sort algorithm we saw earlier

```plaintext
for i = 0 to n-2
  for j = i+1 to n-1
    if A[i] = A[j]
      print i and j
      exit
    continue
```

- i corresponds to your left index finger, j your right index finger
- put your left finger on the first number, your right on the number next to it
- scan right with your right finger until you find a match or the end of the list
- if you reach the end, move your left finger to the right one place and repeat the scan

Matching Pairs in Ruby (cont’d)

```ruby
for i in 0..a.length-2
  for j in i+1..a.length-1
    if a[i] == a[j]
      puts "match: a[#{i}] = #{a[i]}, a[#{j}] = #{a[j]}
      return
    end
  end
end
```

- A test case (after putting this loop in a method named matchpair):

```plaintext
>> a = [86, 63, 39, 98, 96, 38, 68, 88, 83, 17, 33, 69, 66, 89, 96, 93]
>> matchpair(a)
```

#{x} in a string means "insert the value of x here"
Analysis of Matching Pairs

- The goal for algorithm analysis is to come up with an equation for the number of steps as a function of the problem size
  \[ \#\text{steps} = f(n) \]
- The exact number of steps required by the matching pairs algorithm depends on the contents of the arrays
  - could stop after one comparison
  - worst case: no matching pairs, so do all pairwise comparisons
- For this exercise we’ll look at the worst case (no matching pair)
- One way to approach the problem is to draw pictures that show how many times key steps in the algorithm are executed

Number of Steps in Matching Pairs

- From \textit{NTO}: The total number of steps in the worst case is
  \[
  1 + 3(n - 1) + 1 + 3(n - 2) + \cdots + 1 + 3(2) + 1 + 3(1)
  = n + 1 + 3 \sum_{k=1}^{n-1} k
  = n + 1 + \frac{3n(n - 1)}{2}
  = \frac{3n^2 - n + 2}{2}
  \]
  this simplification is an example of what you learn in MTH 231 (you don’t need to understand how we derived this)
- \textbf{Extra credit opportunity:} there is an error here (and one more in the book)
  - it is true that \[ \sum_{k=1}^{n-1} k = n(n - 1)/2 \]
  - send me the correct equation; the best (clearest) will go on the Wiki
Number of Steps in Matching Pairs (cont’d)

- The second visualization shows the number of comparison steps is
  \[ \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \]
- So both ways of counting steps come up with a formula in terms of \( n^2 \)
- Note: the picture shows why if we take away the bottom row we’ll have an \( n \times (n-1) \) matrix with exactly half the cells filled...
- Example: \[ \sum_{k=1}^{5} k = \frac{5 \times 6}{2} = 15 = 1 + 2 + 3 + 4 + 5 \]

Predicting Execution Time

- Ideally we would be able to use the number of steps executed to predict the execution time on a given computer
- In real life, however, lots of different factors affect execution time
  - the “cycle time” of the CPU chip
  - the amount of memory on the machine (an issue for large problems)
  - on Linux, OS/X, and other systems the O/S overhead slows down the program
- But we can still use the formulas from the previous slides as estimates of the order of magnitude of the execution time
  - all steps (NTO): \( (3n^2 - n + 2)/2 \)
  - comparisons only: \( n(n - 1)/2 = n^2 - n \)
  - \( n^2 \) is the dominant term in each case

“Big O” Notation

- The notation computer scientists use to describe the order of magnitude of an algorithm is called “big O”.
- An algorithm is \( O(f(n)) \) if \( f(n) \) is the largest term in the equation for the number of steps
- Here “number of steps” usually means “number of key steps”, such as comparisons
- The matching pairs algorithm is \( O(n^2) \)
  - we also say “matching pairs takes time \( O(n^2) \)”
- When reading this out loud say either “order n-squared” or “oh of n-squared”
- Note: constants are not important
  - both \( (3n^2 - n + 2)/2 \) and \( n^2 - n \) are \( O(n^2) \)

Informal Definition of \( O \)

- Order of magnitude conveys some information about scalability
- The blue line shows the execution time of an algorithm that is \( O(n^2) \)
- The green line is one that is \( O(n) \)
- For small problems the quadratic algorithm might be more efficient
- But eventually at some problem size the linear algorithm will be faster
Classes of Algorithms

- Some common categories of algorithms:
  - $O(1)$: constant
  - $O(\log_2 n)$: logarithmic
  - $O(n)$: linear
  - $O(n \log_2 n)$: "n log n"
  - $O(n^2)$: quadratic
  - $O(n^k)$: polynomial
  - $O(2^n)$: exponential

Analysis of Matching Pairs V.2

- The new algorithm has just one loop instead of nested loops
  
  ```ruby
  for i ← 0 to n-1
    if not seen[A_i]
      seen[A_i] = i
    else
      print i and seen[A_i]
  
  - This algorithm does only $n$ comparisons
  - Running time is $O(n)$
  - But this gain comes at the expense of the amount of space required
  - The number of cells in `seen` must equal the largest number in the input list

Matching Pairs Revisited

- NTO has a second algorithm for the matching pairs problem
  
  ```ruby
  for i ← 0 to n-1
    if not seen[A_i]
      seen[A_i] = i
      print i and seen[A_i]
    else
      # print i and seen[A_i]
  
  - When we first encounter a number
  - Store the location where it was found in the array named ‘seen’

Associative Arrays

- Next problem: how to indicate we have not seen a number yet
  - Common method: use a value that we know isn’t in A
  - e.g. for positive integers in A use -1 to indicate “not seen yet”
  - But then we have another problem: we need a very long loop just to initialize seen
  - Ruby and other languages have a class that solves both these problems
    - Known as an associative array
    - Aka “map” or “hash”
  - Topic for a future lecture: implementation of matching pairs in Ruby, using an associative array
Time vs. Space

- The two algorithms of the matching pairs problems illustrate a very common situation in computer science.
  - most algorithms involve a **tradeoff between time and space**
- It is often possible to lower the execution time by using more space.
- It is often possible to reduce space requirements by increasing the number of steps.
- Big-O notation is also used to describe the amount of storage required.
  - e.g. the first matching pairs algorithm might be described as “quadratic time and linear space.”
  - time: \( O(n^2) \) steps
  - space: \( O(n) \) bytes

Review

- **Skills:**
  - know the major categories of algorithms (linear, n-log-n, etc)
  - given the pseudo-code description of an algorithm like matching-pairs decide which class it belongs to (linear vs quadratic)
  - given the equation for the performance of an algorithm be able to say how many steps it will require for problems of any size

Demo

- The xSortLab applet shows how various sorting algorithms work.

  [xSortLab](http://math.hws.edu/TMCM/java/xSortLab)