“Big O” Notation

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

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>(\log n)</th>
<th>(n \log n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>poker</td>
<td>5</td>
<td>12</td>
<td>2.32</td>
<td>12</td>
</tr>
<tr>
<td>bridge</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 \(n \log n\)

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 and 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
  \[
  for i in j..k ...
  \]
  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

To illustrate the process of analyzing an algorithm we’ll use a very simple technique to see if an array has a matching pair of numbers.

The algorithm scans an array of integers to see if any two numbers are the same.

- if so print the location (index) of the matching items

```plaintext
for i ← 0 to n-2
  for j ← i+1 to n-1
    if A_i = A_j
      print i and j
```

output: 2, 11

Loops in Matching Pairs

This algorithm has nested loops.

- 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

```plaintext
for i ← 0 to n-2
  for j ← i+1 to n-1
    if A_i = A_j
      print i and j
```

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 at #{i}, #{j}"
    end
  end
end
```

Notes:

- i..j is a “range” object -- used in Lab 2 to make arrays
- Ruby variable names start with lower case letters (so it’s a[i] and not A[i])
- puts means “put string”
- the #{i} in the argument to puts means “insert the value of i here”
- Important! The comparison is a[i] == a[j] and not a[i] = a[j]

Q: What happens if we write

```ruby
if a[i] = a[j]
  puts "match at #{i}, #{j}"
```

A test case (after putting this loop in a method named `match_pairs`):

```ruby
>> a = [86,63,39,98,96,38,68,88,36,83,17,33,69,66,89,96,93]
>> match_pairs(a)
machine at 4, 15
```
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)
  - the outer loop will be executed \( n \) times
  - the inner loop on iteration \( i \) will be executed \( n - i \) times

Visualization

- It is often useful to draw pictures that show how many times key steps in the algorithm are executed
- For the matching pairs problem (and other algorithms with nested loops) draw an \( n \times n \) matrix
  - put a dot in the matrix at row \( i \), column \( j \) if the algorithm compares array elements \( A_i \) and \( A_j \)
- This visualization does not account for every single step in the program
- But it does give us some insight into the number of key steps
  - comparing two numbers is the key step in this algorithm -- the rest is “housekeeping”

Number of Steps in Matching Pairs

- The visualization suggests a formula for the number of comparisons:
  \[ \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \]
- This is a basic formula in discrete math, and it can be reduced to
  \[ \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \]
- Example: suppose \( n = 6 \); then
  \[ \sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = \frac{6 \times 5}{2} = 15 \]

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 asymptotic execution time
The Dominant Term

- The first step is to rewrite the formula as a polynomial
  \[ f(n) = \frac{n(n - 1)}{2} = n^2 - n \]
- When written this way it’s apparent that \( n^2 \) is the dominant term in this equation
- As \( n \) gets larger the other terms do not contribute significantly to the value of the formula, i.e. for large enough \( n \)
  \[ f(n) \approx n^2 \]

“Big O” Notation

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

Scalability

- Big O notation tells us something about the scalability of an algorithm
  - it gives us an idea of how the algorithm will behave on larger inputs
- As an example, suppose we have a choice of two algorithms for a problem
  - method A has a nested loop, and the key step inside the inner loop is very simple
  - method B does not have an inner loop, but it requires us to do some other steps before the loop and the key step inside the loop is 20 times more complicated
- We do some algebra to come up with equations that tell us roughly how many steps are required for each algorithm:
  \[ f_A(n) = \frac{n(n - 1)}{2} = n^2 - n \]
  Note algorithm A is \( \mathcal{O}(n^2) \)
  and algorithm B is \( \mathcal{O}(n) \)
  \[ f_B(n) = 100 + 20n \]

Scalability

- When the input size is small (1 ≤ \( n \) ≤ 20)
  algorithm A will require fewer steps
  \[ f_A(n) = \frac{n(n - 1)}{2} = n^2 - n \]
  \[ f_B(n) = 100 + 20n \]
Scalability

- As the input size grows, algorithm B is relatively more efficient.
- Eventually, the $n^2$ term will dominate the count of the number of key steps.

$$f_A(n) = \frac{n(n-1)}{2} = n^2 - n$$

$$f_B(n) = 100 + 20n$$

Classes of Algorithms

- Some common names for major classes 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

Worst Case Behavior

- The actual running time of the matching-pairs algorithm depends on the input data.
  - With good luck, $A_2 = A_1$ and the algorithm does just one comparison.
  - In this case, the algorithm is clearly not $O(n^2)$.
  - Time doesn’t even depend on $n$.
- To be precise, we should say “the worst case asymptotic behavior is $O(n^2)$”
  - Worst case for this algorithm means no matching pairs.
- Other algorithms have the same situation.
  - We’ll look at sorting algorithms later in these slides.
  - Some algorithms are $O(n)$ if the input is sorted already.
Matching Pairs Revisited

- Another way to search for matching pairs use the value of $A_i$ as an index into an array of numbers seen so far
  - before the scan starts make a second array named $loc$
  - initialize $loc[i]$ to $false$ (meaning the value $i$ has not yet been seen in $A$)
- Now the main loop just has to look up $A_i$ in $loc$
  
  ```
  for $i \leftarrow 0$ to $n-1$
  if $loc[A_i] = false$
    $loc[A_i] \leftarrow i$
  else
    print $i$ and $loc[A_i]$
  ```

Analysis of Matching Pairs 2.0

- The new algorithm has just one loop instead of nested loops
  ```
  for $i \leftarrow 0$ to $n-1$
  if $loc[A_i] = false$
    $loc[A_i] \leftarrow i$
  else
    print $i$ and $loc[A_i]$
  ```
- This algorithm does only $n$ comparisons, so the complexity is $O(n)$
- But this gain comes at the expense of the amount of space required
  - maybe a lot of space: the number of cells in $loc$ must equal the largest number in the input list

Analysis of Matching Pairs 2.0

- There is a data structure known as an **associative array** that provides a useful representation for the array named $loc$
  - starts out with 0 elements
  - grows as new elements are added
  - the number of steps to see if $A_i$ is contained in $loc$ is $O(\log_2 m)$
    - where $m$ is the number of items in $loc$
  - The worst case behavior (scan all $n$ elements, find no duplicates) is thus $O(n \log_2 n)$

- Topic for another day: implementation of Matching Pairs 2.0 in Ruby, using Ruby's associative array class (named Hash)

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: $O(n)$ words
Sorting

- Sorting is a very important problem, and there are dozens, if not hundreds, of interesting algorithms
  - Vol. 3 of Knuth is devoted entirely to “Sorting and Searching”
  - almost 400 pages on sorting algorithms
- We’ll finish off this lecture on “Big O” notation by looking at two algorithms
  - insertion sort: $O(n^2)$
  - merge sort: $O(n \log_2 n)$

Insertion Sort

- We’ll use cards as our example
- The goal: sort the cards into ascending order (ignoring suits)
- The basic idea for insertion sort: move through the hand from left to right, and at each location $i$, find the correct placement for card $i$
- Key concept: at the start of each loop iteration, all the cards to the left of location $i$ will be sorted

Insertion Sort (cont’d)

- Here is the pseudo-code for insertion sort
- The cards are in an array named $A$

```java
for i = 1 to n-1
    key ← A_i
    j ← i-1
    while j ≥ 0 and A_j > key:
        A_{j+1} ← A_j
        j ← j-1
    A_{j+1} ← key
```

Demo

- The xSortLab applet shows how various sorting algorithms work

 http://math.hws.edu/TMCM/java/xSortLab
Merge Sort

- The basic idea behind merge sort is **divide and conquer**
- Solve smaller versions of the problem, then combine the partial solutions into the final result
- Divide and conquer is a very important problem solving technique used in many different applications in computer science
- Method:
  - work through the hand two cards at a time, sorting each group of two
  - go through again, **merging** adjacent groups of two into sorted groups of four
  - repeat, merging groups of four into groups of eight...
- This works well because merging is efficient
  - imagine the cards face up on a table
  - one pile for each sorted group, with the lowest card visible on top
  - to merge two piles, just pick the lowest card visible in each group

Merge Sort (cont’d)

- Merge sort uses two arrays
- At each iteration, copy items from one array to the other when doing the merge
- The pseudocode is easier to understand if we break it into two methods
  - msort(a) is the method we call to sort an array named a
  - merge(a,b,i,n) merges a range of elements in a, copying them to b; i is the index of the start of the first group, and n is the size of the group
- Examples:
  - merge(a,b,0,2) means “merge a[0..1] and a[2..3] into b[0..3]”
  - merge(a,b,4,2) means “merge a[4..5] and a[6..7] into b[4..7]”

Merge Sort (cont’d) 

Example with a hand of seven cards

---

**Extra Credit**

```plaintext
msort(a):
  tmp = []
  n = 1
  while n < length(a)
    i = 0
    while i < length(a)
      merge(a,tmp,i,n)
      i += 2*n
    end
    n *= 2
  end
  swap(a,tmp)
  return a
```

---

**Extra Credit**
Merge Sort (cont’d)

\[
\text{merge}(a,b,i,n): \\
\quad \text{lx} = i, \text{rx} = i+n \\
\quad \text{lmax} = \min(\text{length}(a),i+n), \text{rmax} = \min(\text{length}(a),i+2*n) \\
\quad \text{ox} = i \\
\quad \text{while lx < lmax or rx < rmax} \\
\quad \quad \text{if rx} \geq \text{rmax or (lx} < \text{lmax and a[lx] < a[rx])} \\
\quad \quad \quad \text{b[ox]} = \text{a[lx]} \\
\quad \quad \quad \text{lx} += 1 \\
\quad \quad \text{else} \\
\quad \quad \quad \text{b[ox]} = \text{a[rx]} \\
\quad \quad \quad \text{rx} += 1 \\
\quad \quad \text{ox} += 1 \\
\quad \text{end}
\]

Extra Credit

Merge Sort Demo

\[
> \text{load "msort.rb"} \\
> \text{a} = [] \\
> \Rightarrow [] \\
> 8.\text{times} \{ \text{a} \ll \text{rand}(100) \} \\
> \Rightarrow [] \\
> \text{msort(a, :trace)} \\
\Rightarrow [75, 52, 94, 36, 2, 79, 31, 84] \\
\Rightarrow \text{a} \\
\Rightarrow [75, 52, 94, 36, 2, 79, 31, 84] \\
\Rightarrow [52, 75, 36, 94, 2, 79, 31, 84] \\
\Rightarrow [36, 52, 75, 94] \\
\Rightarrow [2, 31, 79, 84] \\
\Rightarrow [2, 31, 36, 52, 75, 79, 84, 94]
\]

Complexity of Merge Sort

- To see why merge sort is $O(n \log_2 n)$ note that at each iteration of the outer loop we have to look at all $n$ cards
- But the size of the piles doubles each time, so the total number of iterations is $O(\log_2 n)$
- Try it!
  - put all 52 cards face up
  - merge groups of 1, then groups of 2, ...
  - keep track of number of comparisons
  - do it again with insertion sort
  - should be about 300 comparisons vs 1200 for insertion sort
  - how much time does each sort take?

Demo

- The xSortLab applet includes merge sort as one of its demos
  
  http://math.hws.edu/TMCM/java/xSortLab
Review

Skills:

- know the major categories of algorithms (linear, n-log-n, etc)
- given the pseudo-code description of an algorithm like matching-pairs:
  - count the number of operations (e.g. number of comparisons)
  - decide which class it belongs to (linear vs quadratic)
  - single loop: most likely $O(n)$
  - nested loops: could be $O(n^2)$ or $O(n \log n)$
    depending on how many times outer loop is executed
- given the equation for the performance of an algorithm be able to say how many steps it will require for problems of any size
- don’t worry about deriving and/or simplifying equations