“For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing.”
– Francis Sullivan

What is an algorithm?

Examples

– sort a list of numbers
– find a route from one place to another (cars, packet routing, phone routing, ...)
– find the longest common substring between two strings
– add two numbers
– microchip wiring/design (VLSI)
– solving sudoku
– cryptography
– compression (file, audio, video)
– spell checking
– pagerank
– classify a web page
– ...

What properties of algorithms are we interested in?

– does it terminate?
– is it correct, i.e. does it do what we think it’s supposed to do?
– what are the computational costs?
– what are the memory/space costs?
- what happens to the above with different inputs?
- how difficult is it to implement and implement correctly?

• Why are we interested? Most of the algorithms/data structure we will discuss have been around for a while and are implemented. Why should we study them?

  - For example, look at the java.util package
    * Hashtable
    * LinkedList
    * Stack
    * TreeSet
    * Arrays.binarySearch
    * Arrays.sort
  - Know what’s out there/possible/impossible
  - Know the right algorithm to use
  - Tools for analyzing new algorithms
  - Tools for developing new algorithms
  - interview questions? :) 

    * Describe the algorithm for a depth-first graph traversal.
    * Write a function f(a, b) which takes two character string arguments and returns a string containing only the characters found in both strings in the order of a. Write a version which is O(n^2) and one which is O(n).
    * You’re given an array containing both positive and negative integers and required to find the sub-array with the largest sum (O(n) a la KBL). Write a routine in C for the above.
    * Reverse a linked list
    * Insert in a sorted list
    * Write a function to find the depth of a binary tree
    * ... 

  - Personal experience: Understanding and developing new algorithms has been one of the most useful tools/skills for me.
    * Hierarchical clustering
    * Perceptron learning algorithm
    * Sparse vector manipulation
• Pseudocode

- A way to discuss how an algorithm works that is language agnostic and without being encumbered with actual implementation details.
- Should give enough detail for a person to understand, analyze and implement the algorithm.
- Conventions

Mystery1($A$)
1 $x \leftarrow -\infty$
2 for $i \leftarrow 1$ to $\text{length}[A]$ \\
3 if $A[i] > x$ \\
4 $x \leftarrow A[i]$
5 return $x$

Mystery2($A$)
1 for $i \leftarrow 1$ to $\lfloor \text{length}(A)/2 \rfloor$
2 swap $A[i]$ and $A[\text{length}(A) - (i - 1)]$

- Comments
  * array indices start at 1 not 0
  * we may use notation such as $\infty$, which, when translated to code, would be something like Integer.MAX_VALUE
  * use shortcuts for simple function (e.g. swap) to make pseudocode simpler
  * we’ll use $\leftarrow$ instead of $=$ to avoid ambiguity
  * Indentation specifies scope

• Sorting

Input: An array of numbers $A$
Output: The array of numbers in sorted order, i.e. $A[i] \leq A[j] \forall i < j$

- cards
* sort cards: all cards in view
* sort cards: only view one card at a time

- Insertion sort

**Insertion-Sort**

1. \textbf{for} \( j \leftarrow 2 \) \textbf{to} \( \text{length}[A] \)
2. \hspace{1em} \( \text{current} \leftarrow A[j] \)
3. \hspace{1em} \( i \leftarrow j - 1 \)
4. \hspace{1em} \textbf{while} \( i > 0 \) \text{ and } A[i] > \text{current} \)
5. \hspace{2em} \( A[i + 1] \leftarrow A[i] \)
6. \hspace{2em} \( i \leftarrow i - 1 \)
7. \hspace{1em} \( A[i + 1] \leftarrow \text{current} \)

- Does it terminate?
- Is the algorithm correct?

Loop invariant: A statement about the algorithm that is always true regardless of where we are in the algorithm

**Insertion-Sort** invariant: At the start of each iteration of the \textbf{for} loop of lines 1-7 the subarray \( A[1..j-1] \) is the sorted version of the original elements of \( A[1..j-1] \)

To prove, need to show two things:
* Base case: invariant is true before the loop
* Inductive case: it is true after each iteration

upon termination of the loop, the invariant should help you show something useful about the algorithm.

Proof

- Running time: How long does it take? How many computational "steps" will be executed?

What is our computational model? Turing machine? We'll assume a random-access machine (RAM) model of computation.

Examine costs for each step
\[ T(n) = c_1 n + c_2 (n-1) + c_3 \sum_{j=2}^{n} t_j + c_4 \sum_{j=2}^{n} (t_j - 1) + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 (n - 1) \]

* Best case: array is sorted
  \[ t_j = 1 \]
  \[ \sum_{j=2}^{n} = n - \text{Linear} \]
* Worst case: array is in reverse sorted order
  \[ t_j = j \]
  \[ \sum_{j=2}^{n} = n + n - 1 + n - 2 + \cdots + 2 = \frac{n(n+1)}{2} - 1 - \text{Quadratic} \]
* Average case: array is in random order
  The array up through \( j \) is sorted. How many entries on average will we have to analyze before in the sorted portion of the array to find the correct location for the current element?
  \[ t_j = j/2 \]
  \[ \sum_{j=2}^{n} = \frac{n(n+1)}{2} - 1/2 - \text{Quadratic} \]
* Can we do better? What about if we used binary search to find the correct position?

- **Divide and Conquer**
  - *Divide* the problem into smaller subproblems
  - *Conquer* the subproblems by solving the subproblems. Often this just involves waiting until the problem is small enough that it is trivial to solve.
  - *Combine* the divided subproblems into a final solution.

**Merge-Sort(A)**

```
1   if length[A] == 1
2       return A
3 else
4       q ← ⌊length[A]/2⌋
5       create arrays L[1..q] and R[q+1..length[A]]
6       copy A[1..q] to L
7       copy A[q+1..length[A]] to R
8       LS ← Merge-Sort(L)
9       RS ← Merge-Sort(R)
10      return Merge(LS, RS)
```
Merge($L, R$)
1. create array $B$ of length $\text{length}[L] + \text{length}[R]$
2. $i \leftarrow 1$
3. $j \leftarrow 1$
4. for $k \leftarrow 1$ to $\text{length}[B]$
5.     if $j > \text{length}[R]$ or $(i \leq \text{length}[L] \text{ and } L[i] \leq R[j])$
6.         $B[k] \leftarrow L[i]$
7.         $i \leftarrow i + 1$
8.     else
9.         $B[k] \leftarrow R[j]$
10.        $j \leftarrow j + 1$
5. return $B$

– Is the algorithm correct?

Merge invariant: At the end of each iteration of the for loop of lines 4-10 the subarray $B[1..k]$ contains the smallest $k$ elements from $L$ and $R$ in sorted order.

Proof?

– Running time

$$T(n) = \begin{cases} 
    c & \text{if } n \text{ is small} \\
    2T(n/2) + D(n) + C(n) & \text{otherwise}
\end{cases}$$

$D(n)$ Divide: copy the input array into two halves - linear, $\Theta(n)$
$C(n)$ Combine: merges the two sorted halves - linear, $\Theta(n)$

$$T(n) = \begin{cases} 
    c & \text{if } n \text{ is small} \\
    T(n/2) + cn & \text{otherwise}
\end{cases}$$

Analyze the tree on pg. 35
$cn \log n + cn$

Merge-Sort2($A, p, r$)
1. if $p < r$
2.     $q \leftarrow \lfloor (p+r)/2 \rfloor$
3.     Merge-Sort2($A, p, q$)
4.     Merge-Sort2($A, q + 1, r$)
5.     Merge2($A, p, q, r$)
\textbf{MERGE2}(A, p, q, r)\\
1 \hspace{0.5cm} n_1 \leftarrow q - p + 1 \hspace{0.5cm} \triangleright \text{ length of the left array}\\
2 \hspace{0.5cm} n_2 \leftarrow r - q \hspace{0.5cm} \triangleright \text{ length of the right array}\\
3 \hspace{0.5cm} \text{create arrays } L[1..n_1 + 1] \text{ and } R[1..n_2 + 1]\\
4 \hspace{0.5cm} \textbf{for } i \leftarrow 1 \textbf{ to } n_1\\
5 \hspace{1cm} L[i] \leftarrow A[p + i - 1]\\
6 \hspace{0.5cm} \textbf{for } j \leftarrow 1 \textbf{ to } n_2\\
7 \hspace{1cm} R[j] \leftarrow A[q + j]\\
8 \hspace{0.5cm} L[n_1 + 1] \leftarrow \infty\\
9 \hspace{0.5cm} R[n_2 + 1] \leftarrow \infty\\
10 \hspace{0.5cm} i \leftarrow 1\\
11 \hspace{0.5cm} j \leftarrow 1\\
12 \hspace{0.5cm} \textbf{for } k \leftarrow p \textbf{ to } r\\
13 \hspace{1cm} \textbf{if } L[i] \leq R[j]\\
14 \hspace{1.5cm} A[k] \leftarrow L[i]\\
15 \hspace{1.5cm} i \leftarrow i + 1\\
16 \hspace{1cm} \textbf{else}\\
17 \hspace{1.5cm} A[k] \leftarrow R[j]\\
18 \hspace{1.5cm} j \leftarrow j + 1\\

– Is the algorithm correct?\\
– Running time\\
\hspace{0.5cm} \text{Same as } \text{MERGE-SORT except } D(n) = c\\

This still results in:\n\[ T(n) = 2T(n/2) + cn \]

– What are the memory/space costs of the two merge sort algorithms?\\
\hspace{0.5cm} \text{Memory usage is different than time usage: we can reuse memory!}\\
\hspace{0.5cm} \text{In general, we’re interested in maximum memory usage, but may also be interested in average memory usage while processing.}\\
– How hard are the two merge sort versions to implement/debug?\\

• Bubble sort
Bubble-Sort(A)
1  sorted ← false
2  while sorted = false
3      sorted ← true
4        for i ← 1 to length[A] − 1
5        if A[i] > A[i + 1]
6          swap A[i] and A[i + 1]
7          sorted ← false

These notes are adapted from material found in chapters 1 + 2 of [1].

References