W1005
Intro to CS and Programming in MATLAB

Algorithms

Fall 2014
Instructor: Ilia Vovsha

http://www.cs.columbia.edu/~vovsha/w1005

Algorithms (part 1)

- Selection sort algorithm
  - Subfunctions
  - Scope of variables
- Running time, big-O notation
  - for loops, if-else statements
  - Short-circuiting
- Insertion sort algorithm
  - Comparison of algorithms
  - Function handles

Algorithms

- General considerations:
  - Set of rules to accomplish a specific task
  - Process information into ideal format (e.g. sort before displaying)
  - Running time, resources (memory)
  - Average/worst/best case scenarios
  - Possible approach: break down complicated problem into simpler sub-problems. Solve sub-problems with existing, tested components
  - When do we terminate the algorithm?
  - How do we compare algorithms?

Selection Sort

- Pseudocode:
  - **Input**: vector V with N elements
  - **Goal**: sort vector in ascending order
  - **Assumption**: comparison based sorting (> < operators exist)
  - **Algorithm**:
    1. Set k = 1
    2. Locate minimum element in (sub)vector V(k..N)
    3. Switch (swap) that element with element at index k
    4. Increment k (k = k+1) and go to step 2, stop when k = N-1

Selection Sort (code)

```
function  V = ssort(V)
N = length(V);

for  k = 1:N-1
     % locate minimum index
     idx = locate_min(V,k,N);
     % swap elements
     V = swap(V,idx,k);
end
```

Subfunctions

- Typically one function for one file (file name = function name)
- Also possible to include multiple functions in one file:
  - Main function and sub-functions
  - Sub-functions are only “visible” to other functions in the same file
  - Useful when you have many minor and very specific routines
  - Type ‘help function’ to see an example

Selection Sort (subfunctions)

```
function  V = ssort(V)
N = length(V);
for  k = 1:N-1
     idx = locate_min(V,k,N);
     V = swap(V,idx,k);
end

function idx = locate_min(V,k,N)
% body of subfunction
function V = swap(V,E1,E2)
% body of subfunction
```

Scope

- Scope of name: region where particular meaning of a name is visible or can be referenced
- Each function has a separate "workspace" (scope), but they can share variables if they are declared *global* within each desired scope
    - global my_cell
- Selection sort example:
    - Add: global V declaration to main function and "swap" subfunction
- NOT recommended! (alternative: nested functions)

Selection sort (analysis)

- Estimate efficiency of sorting algorithm:
  - Number of element comparisons
  - Number of element exchanges
- Selection sort:
  - First iteration of the loop: N-1 comparisons, 1 exchange
  - Some iteration: N-k comparisons, 1 exchange
  - Last iteration: 1 comparison, 1 exchange
  - Total comparisons: (N-1) + (N-2) + …..+ 2 + 1   = N x (N-1) / 2
  - Total exchanges: N-1 (at most)

Big-O Notation

- Estimate efficiency of algorithm, relative to other algorithms for identical task
  - Difficult to get precise measure
  - Approximate effect on change of number of items (n) processed
  - Compare growth rates
  - Order of magnitude (O)
- Definitions:
  - T(N) = **O**(*f(N)*) ➔ {exist c,n_0 > 0}: T(N) ≤ c*f(N) when N ≥ n_0*
  - T(N) = **Ω**(*f(N)*) ➔ {exist c,n_0 > 0}: T(N) ≥ c*f(N) when N ≥ n_0*
  - T(N) = **θ**(*f(N)*) ⟷ T(N) = **O**(*f(N)*) and T(N) = **Ω**(*f(N)*)

Big-O Notation (rules)

- Conventions:
  - T(N) = **O**(*f(N)*) ➔ *f* upper bound on T
  - T(N) = **Ω**(*f(N)*) ➔ *f* lower bound on T
  - Choose tightest bound
  - Don't include constants, or lower order terms
  - Using L'Hopital's rule is usually overkill
- Rules: T1(N) = **O**(*f(N)*), T2(N) = **O**(*g(N)*)
  a) T1(N) + T2(N) = max { **O**(*f(N)*), **O**(*g(N)*) }
  b) T1(N) * T2(N) = **O**(*f(N)* * *g(N)*)

Running Time Calculations

- Estimate running time of algorithm:
  - Worst case scenario: bound
  - Average case scenario: hard to compute
  - Analysis pinpoints bottlenecks
  - No particular units of time
  - Analyze inside out
- General rules (worst case)
  - **For Loops**: RT(statements inside loop) x (# iterations)
  - **Nested Loops**: RT(inner for loop) x (sizes of outer loops)
  - **Consecutive Statements**: add RT (maximum counts)
  - **If/Else**: RT(condition test) + max{RT(if code block),RT(else code block)}

Logical Short-Circuiting

- Logical operators for conditional statements: with logical short-circuiting, the second operand is evaluated only when the result is not fully determined by the first operand
- &&, ||
  - logical operation with short-circuiting behavior
  - Each expression must evaluate to a scalar logical result
  - Using the & and | operators for short-circuiting can yield unexpected results when the expressions do not evaluate to logical scalars

Insertion Sort

- Pseudocode:
  - **Input**: vector V with N elements
  - **Goal**: sort vector in ascending order
  - **Assumption**: comparison based sorting (> < operators exist)
  - **Algorithm**:
    1. Set k = 2
    2. Sort (sub)vector V(1..k)
       - ➔ Using fact that subvector V(1..k-1) is already sorted
       - ➔ Move element in position k left until correct place found
    3. Increment k (k = k+1) and go to step 2, stop when k = N

Insertion Sort (code)

```
function  V = isort(V)
N = length(V);

for  k = 2:N
      % find where to move left
      idx = move_left(V,k);
      % swap elements
      V = swap(V,idx,k);
end
```

Insertion Sort (example)

34	8	64	51	32	21	Original
8	34	64	51	32	21	After k=2
8	34	64	51	32	21	After k=3
8	34	51	64	32	21	After k=4
8	32	34	51	64	21	After k=5
8	21	32	34	51	64	After k=6

Function Handles @

- Function handle: a variable that stores an identifier for a function
  - h1 = @min; % ‘h1’ can now be used instead of ‘min’
  - val = h1(rand(3)) % Same as val = min(rand(3))
- Can use handles in cell arrays or structs, but not in regular arrays:
  - C = {@min, @max, @mean};
  - S.a = @min; S.b = @max; S.c = @mean;
  - ~~A = [@min,@max];~~ WRONG!

Function Handles @

- Purpose of function handles:
  - Suppose the user should be able to choose which sub-routine to use. Thus, a mechanism to pass a parameter which specifies the sub-routine is required
  - Example: which sorting algorithm to use
  - Can’t pass a function, so pass a handle instead
  - Also possible to define functions on the fly (*anonymous* functions):
    1. sqr = @(x) x.^2;
    2. a = sqr(5);

Algorithms (part 2)

- Binary Search
- Mergesort algorithm
  - Debugging, time functions
  - Profiling
- Running time for recursive functions
  - Fibonacci numbers
  - P vs NP

Binary Search

- Problem definition:
  - Given sorted list (vector) of numbers V, and a number X. Find the index k such that V(k) = X. Return k = -1 if X is not in V.
- Solution approach:
  - Scan through the list and compare X to each element
  - Linear running time, does not take advantage of the list being sorted
- Better approach:
  - *Divide and conquer* algorithms
  - Break problem into subproblems of the same type
  - Constant time to cut problem size by a fraction (usually ½)

Binary Search (algorithm)

- Pseudocode:
  - **Input**: *sorted* (ascending order) vector V with N elements, element X
  - **Goal**: find index k where V(k) = X or return -1
  - **Assumption**: > < operators exist
  - **Algorithm**:

```
1.  low_k = 1, high_k = N
2.  while low_k <= high_k           % we still have indices to check
        m = (low_k + high_k) / 2    % middle element
        compare X to V(m)
        If X == V(m) ➔ stop, return k = m
        If X > V(m) ➔ search right half: low_k = m + 1;
        if X < V(m) ➔ search left half: high_k = m – 1;
3.  Return k = -1                   % we failed to find X in V
```

Binary Search (analysis)

- Binary search is a recursive algorithm, we can define and solve a *recurrence relation*:
  - Base case: T(0) = constant
  - Recursive case: T(N) = T(subproblems) + T(combine solutions)
- Running time depends on:
  - Number of subproblems
  - Size of subproblems
  - Cost of combining solutions

Mergesort (idea)

- Classic divide and conquer strategy:
  - Divide list into 2 halves (each half = subproblem)
  - Apply algorithm recursively to sort each half
  - Merge the two sorted lists
- Merging two sorted lists:
  - One pass through the input (N elements)
  - Linear running time, at most N-1 comparisons
  - Requires a temporary array (additional resource)

Mergesort (algorithm)

- Pseudocode:
  - **Input**: vector V with N elements
  - **Goal**: sort vector in ascending order
  - **Assumption**: comparison based sorting (> < operators exist)
  - **Algorithm**:

```
1.  left = 1, right = N
2.  if  left < right                    % we still need to sort
        m  =  (left + right) / 2        % middle element
        mergeSort(V, left, m, T)        ➔ sort left half
        mergeSort(V, m+1, right, T)     ➔ sort right half
        merge(V, left, m+1, right, T)   ➔ merge sorted halves
3.  Return V                            % finished
```

Mergesort (example)

34	8	64	51	32	21	99	3	Original
34	8	64	51	32	21	99	3	[split][sort L]
34	8	64	51	32	21	99	3	[split][sort LL]
8	34	64	51	32	21	99	3	merge(LL)
								sort(LR)
8	34	51	64	32	21	99	3	merge(LR)
8	34	51	64	32	21	99	3	merge(L)
8	34	51	64	32	21	99	3	sort(R)
8	34	51	64	32	21	99	3	[split][sort RL]
8	34	51	64	21	32	99	3	merge(RL)
								sort(RR)
8	34	51	64	21	32	3	99	merge(RR)

Mergesort (code)

```
function  SV = msort(V)
N = length(V);
if  N == 1
      SV = V;
else
      m = floor(N/2);              % Middle element
      VL = msort(V(1:m));          % Sort left half
      VR = msort(V(m+1:N));        % Sort right half
      SV = merge(VL,VR);           % Merge
end

function M = merge(L,R)
% body of subfunction
```

Time Functions

- Time a sequence of operations:
  - Start stopwatch: tic
  - Stop stopwatch, display elapsed time: toc
  - Related functions: clock, etime, cputime
  - To track time progress include multiple ‘toc’ commands (but a single ‘tic’ command)
  - Example: tic; SV = msort(V); toc; SV
  - ‘tic’ / ‘toc’ inside ‘msort’ → different purpose (due to recursive calls)

Profiling

- Optimize execution of M-files:
  - Turn it on: profile on
  - Run the code / functions
  - View generated report in HTML format: profile viewer
  - Create report without GUI: profsave(profile('info'), 'dir_name')
- Helpful functions for Debugging:
  - Give control to keyboard (user): keyboard
  - Parse file for syntax errors: mlint ftest
  - 'doc debug'

Binary Search (solve recurrence)

- Binary search is a recursive algorithm, we can define and solve a *recurrence relation*:

$$T(1) = C$$
$$T(N) = T(N/2) + C \quad N > 1$$

$$T(N) = T(N/4) + C + C$$
$$T(N) = T(N/8) + 3C$$
$$T(N) = T(N/2^k) + k*C$$
$$T(N) = T(1) + C*\log N$$
$$T(N) = O(\log N)$$

Factorial (analysis)

- What is the running time? Just a thinly veiled “for loop”, hence O(N)
- Poor use of recursion (easy to convert into a loop)

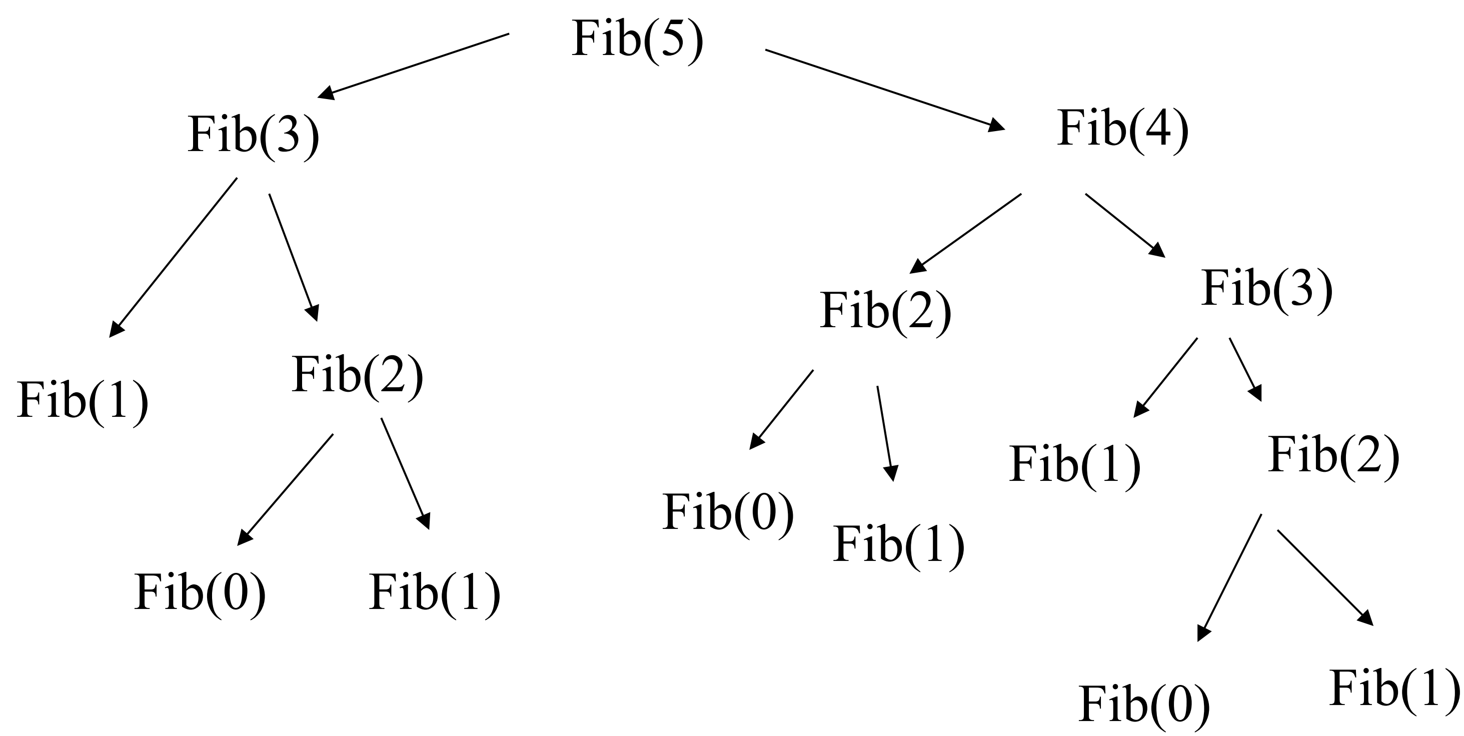
```
function  v = recfact(n)
if  n  <=  1
      v = 1;
else
      v = n * recfact(n-1);
end
```

Fibonacci Numbers (code)

- $F_0 = 1$, $F_1 = 1$, $F_2 = 2$, $F_3 = 3$, $F_4 = 5$,…, $\mathbf{F_i = F_{i-1} + F_{i-2}}$
- Clever use of recursion?

```
function  F = fib(N)
if  N  <=  1
      F = 1;
else
      F = fib(N-1) + fib(N-2);
end
```

Fibonacci Function (call tree)



Fibonacci Numbers (analysis)

- $F_0 = 1, F_1 = 1, F_2 = 2, F_3 = 3, F_4 = 5, \ldots, \mathbf{F_i = F_{i-1} + F_{i-2}}$
- Running time: **T(N) = fib(N)**

$T(0) = T(1) = C$	$N = 0, N = 1$
$T(N) = T(N-1) + T(N-2) + 2$	$N > 1$
$T(N) = F_N < (5/3)^N$	
$T(N) = F_N \geq (3/2)^N$	$N > 4$

- Proof by induction:
  - Demonstrate simple (base) case
  - Assume inductive hypothesis is correct up to some k
  - Prove statement for k+1

Proof by Induction

- $F_0 = 1$, $F_1 = 1$, $F_2 = 2$, $F_3 = 3$, $F_4 = 5, \ldots,$ $\mathbf{F_i = F_{i-1} + F_{i-2}}$

Given: $F_N = F_{N-1} + F_{N-2}$

Prove: $F_N < (5/3)^N$

$$
\begin{aligned}
F_N &= F_{N-1} + F_{N-2} \\
&< (5/3)^{N-1} + (5/3)^{N-2} \\
&< (3/5)(5/3)^N + (3/5)^2(5/3)^N \\
&< (3/5+9/25)(5/3)^N \\
&< (24/25)(5/3)^N \\
&< (5/3)^N
\end{aligned}
$$

Fibonacci Numbers (summary)

- $F_0 = 1$, $F_1 = 1$, $F_2 = 2$, $F_3 = 3$, $F_4 = 5$,…, $\mathbf{F_i = F_{i-1} + F_{i-2}}$

$$T(0) = T(1) = C \qquad N = 0, N = 1$$
$$T(N) = T(N-1) + T(N-2) + 2 \qquad N > 1$$

$$T(N) = F_N < (5/3)^N$$
$$T(N) = F_N \geq (3/2)^N \qquad N > 4$$
$$T(N) = \mathbf{O}((5/3)^N)$$
$$T(N) = \mathbf{\Omega}((3/2)^N)$$
$$T(N) = \mathbf{\theta}(r^N), \quad \text{where } r = (1 + \sqrt{5})/2$$

- Exponential, very inefficient!
- Don't throw away work!

Mergesort (analysis)

- Similar to Binary Search, we can define and solve a *recurrence relation*:

$$T(1) = C$$
$$T(N) = 2T(N/2) + N \qquad N > 1$$

$$T(N/2) = 2T(N/4) + N/2$$
$$2T(N/2) = 2[\ 2T(N/4) + N/2\ ] = 4T(N/4) + N$$
$$\mathbf{T(N) = 4T(N/4) + 2N}$$
$$4T(N/4) = 8T(N/8) + N$$
$$\mathbf{T(N) = 8T(N/8) + 3N}$$
$$\mathbf{T(N) = 2^kT(N/2^k) + k*N}$$
$$T(N) = NT(1) + N \log N \qquad N = 2^k$$
$$T(N) = O(N \log N)$$

P vs NP

- "P": class of problems which can be solved with polynomial time algorithms
- "NP": (nondeterministic polynomial time, exponential) class of problems whose solution can be verified in polynomial time
- Implications of P = NP:
  - Complete chaos
  - Can solve problems as quickly as we can verify the solution
  - Cryptography breaks
  - Mathematicians replaced by machines

Algorithms (part 3)

- Quicksort algorithm
  - Picking the pivot
  - Partitioning strategy

Quicksort (idea)

- Divide and conquer recursive algorithm:
  - Pick some “pivot” element “x” in the original list (vector)
  - Partition the remaining elements in the list “V – {x}” into two disjoint groups: { elements less than x, elements greater than x }
  - Apply algorithm recursively to sort each group
  - Return sorted list: “group 1” then “x” then “group 2”
- Partition step:
  - What should we do with elements equal to the pivot?
  - Linear running time, at most N-1 comparisons
  - Why better than mergesort?

Quicksort (running time)

- Running time:
  - Like mergesort, quicksort solves two subproblems and the combination step requires linear time
  - Issue: subproblems not guaranteed to be of equal size!
  - Worst case running time is $O(N^2)$
  - Can ensure worst case is highly unlikely with proper strategy to pick pivot
  - Partition step can be performed in place (efficient, does not require an additional vector)

Quicksort (picking the pivot)

- Simple (but not recommended) approach:
  - Choose first element of vector as the pivot
  - Works fine if input is random (order)
  - Quadratic running time if input is presorted!
- Better approach:
  - Good partition means subproblems are of close to equal size
  - Choose random element of vector as pivot (safe)
  - Compute median of {first, middle, last} elements of vector

Quicksort (example)

34 8 64 45 51 32 21 99 3	Original
34 8 64 45 51 32 21 99 3	Select pivot
34 8 64 45 51 32 21 99 3	Partition
[8 32 21 3] 34 [64 45 51 99]	Recursive call
[3 8 21 32] 34 [45 51 64 99]	Return

Quicksort (partitioning strategy)

- Purpose:
  - We have picked some "pivot" element: X = V(p)
  - Partition the remaining elements into two disjoint groups {G1, G2}
  - What should we do with elements equal to the pivot?
- Strategy:
  - Get pivot out of the way: swap pivot with last element
  - Track two pointers {L = 1, R = N-1} until they cross (R < L)
  - While V(L) < X, increment L. While V(R) > X, decrement R.
  - Push large elements right and small elements left when you stop: If L < R, swap V(L) & V(R)
  - When pointers cross: swap pivot with V(L)

Partitioning Strategy (example)

34 8 64 45 51 32 21 99 3	Original
34 8 64 45 51 32 21 99 3	Select pivot
3 8 64 45 51 32 21 99 34	Swap pivot & last
3 8 64 45 51 32 21 99 34	Track: L = 1, R = 8
3 8 64 45 51 32 21 99 34	Stop: L = 3, R = 7
3 8 21 45 51 32 64 99 34	Swap: L = 3, R = 7

Partitioning Strategy (example)

34	8	64	45	51	32	21	99	3	Original
3	8	21	45	51	32	64	99	34	Swap: L = 3, R = 7
3	8	21	45	51	32	64	99	34	Stop: L = 4, R = 6
3	8	21	32	51	45	64	99	34	Swap: L = 4, R = 6
3	8	21	32	51	45	64	99	34	Stop: L = 5, R = 4
3	8	21	32	34	45	64	99	51	Swap pivot back

Strategy (equal elements)

- Increment pointer?
  - Pointer “L” should behave as pointer “R” (stop or don’t stop)
  - To ensure O(N logN) running time, must create nearly equal subproblems (groups)
  - Consider a list of N identical elements, which approach (stop or don’t stop) is better?
  - Unnecessary swaps cost less than very uneven groups
  - Best strategy: stop when pivot is equal to current element (left or right)

Quicksort (algorithm)

- Pseudocode:
  - **Input**: vector V with N elements
  - **Goal**: sort vector in ascending order
  - **Assumption**: comparison based sorting (> < operators exist)
  - **Algorithm**:

```
1.  lo = 1, hi = N
2.  if  lo  <  hi                    % we still need to sort
        p  =  pick_pivot(V, lo, hi)      % pick pivot
        np  =  partition(V, lo, hi, p)   % partition strategy
        quicksort(V, lo, np-1)           → sort group 1 (small)
        quicksort(V, np+1, hi)           → sort group 2 (large)
3.  Return V                         % finished
```