Introduction to algorithms
The set-intersection problem

Lecture 2

Algorithms

- A **systematic** and **unambiguous** procedure that produces - in a **finite number of steps** - the answer to a question or the solution of a problem
- An algorithm has an input:
  - Example ?
  - Sometimes, the algorithm works only if the input satisfies some conditions (pre-conditions). **These need to be specified clearly!!!**
    - Examples?
- An algorithm has an output:
  - The solution to the problem (hopefully!)
    - Examples?

What is a *good* algorithms?

- Correctness:
  - Ideally: always returns the right answer
  - When the problem is too hard, we want the algo to
    - Return the right answer most of the time, or
    - Returns an answer that is guaranteed to be close to the right answer
- Speed: Time it takes to solve the problem
- Space: Amount of memory required
- Simplicity:
  - Easy to understand and analyze
  - Easy to implement
  - Easy to debug, modify, update

Running time

- How to measure the speed of an algorithm?
- Problem #1: Running time depends on the size of input.
  - intersecting two big lists takes more time than two small ones
- Solution:
  - Describe running time as a function of input size

Examples:

To compute the average of a set of n numbers, the running time may be $T(n) = 123*n + 0.3$ microseconds.

To compute the intersection of a list of m students with a list of n students, the running time may be

$T(m, n) = 234\ m\ (n \log(n) + 53 \log(n) + 123)$

Running time (2)

Problem #2:

Running time depends not only on the *size* of the input but sometimes also on the *content* of the input itself (called the instance of the problem)

Example: For the list-intersection problem, an algorithm may be fast on

(Alice, Bob, Carl, Don) vs (Alice, Bob, Carl, Don)

but it may be slow on

(Alice, Bob, Carl, Don) vs (Don, Carl, Bob, Alice)

Running time (3)

Three possibilities to measure running time

- Best case: Time on the easiest input of fixed size.
  - Usually meaningless
- Average case: Time on average input
  - Good measure, but very hard to calculate.
  - "Average" according to what input distribution?
- Worst case: Time on most difficult input
  - Good for safety critical systems: airplane traffic control
  - Easier to estimate

Languages for describing algorithms

- English Prose:

  To find the maximum element of an array, initialize m to the value of the first element. Then, for each subsequent element, if that element is larger than m, replace m with the value of that element. Return the value of m.

- Binary language:

  0101010110110011010101001101010010101010101001 0101011101100101010101101010101010010101010101 0010101010010110

- Programming language (Java, C…)

```
int findMax(int A[], int n) {
  int m=A[0];
  for (int i=1;i<n;i++)
     if (m<A[i]) m=A[i]
return m;
}
```

+ Human readable
- Too vague
- Too verbose

+ Very precise
- Human unreadable

+ Precise,
+- Human readable
– Requires knowledge of PL used
- Requires implementation

Pseudo-code

- Universal language to describe algorithms to human, independent of the programming language
- Has common constructs like
  - Assignments: $x \leftarrow x+1$
  - Conditionals: **if** (x=0) **then** …
  - Loops: **for** $i \leftarrow 0$ **to** n **do** …
  - Objects and function calls: triangle.getArea()
  - Mathematical notation: $x \leftarrow \lfloor y^3/2 \rfloor$
  - Blocks, indicated by indentation

Algorithm findMax(A, n)
Input: An array A of n numbers
Output: The largest element of the array
m ← A[0]
for i ← 1 **to** n-1 **do** {
  if (m < A[i]) **then** m ← A[i]
}
return m

List-intersection problem

- Input:
  - The names of a set of students taking COMP250
  - The names of a set of students taking MATH240.
  - **Assumption:** No two students have the same name
- Question:
  - How many students are taking both classes?
- How do I minimize the number of times I need to compare two names?

Solution 1 –Nested for-loops

Algorithm ListIntersection(A,m, B,n)
Input: An array A of m strings and an array B of m strings. The elements of A and B are assumed to be distinct.
Output: The number of elements present in both A and B
inter ← 0
for i ←0 **to** m-1 **do** {
  for j ← 0 **to** n-1 **do** {
    if (A[i] = B[j]) **then** {
      inter ← inter + 1
    }
  }
}
return inter

Solution 2 – Binary search

Algorithm listIntersection(A,m, B,n)
Input: same as before
Output: same as before
inter ← 0
B ← sort (B,n)
for i ← 0 **to** m-1 **do** {
  if (binarySearch(B, n, A[i])) **then** {
    inter ← inter+1
  }
}
return inter

Algorithm sort(A,n)
Input: An array A of n elements.
Output: The array sorted in increasing order
[Assumed to be given]

Algorithm binarySearch(A,n, k)
Input: A *sorted* array A of n elements. Key k
Output: True iif A contains k
left ← 0
right ← n
while (right > left+1) **do** {
  mid ← ⌈(left+right)/2⌉
  if (A[mid]>k) **then** right ← mid
  else left ← mid
}
if (A[left] = k) **then return** True;
else return False;

Solution 2 – Binary search

```
Algorithm sort(A,n)
Input: An array A of n elements.
Output: The array sorted in increasing order
[Assumed to be given]
```

Number of comparisons: ~ $\lceil n \log_2(n) \rceil$

We'll see why next month...

```
Algorithm binarySearch(A,n, k)
Input: A sorted array A of n elements. A key k
Output: True if A contains k, False otherwise
left ← 0
right ← n
while (right > left+1) do
   mid ← ⌈(left+right)/2⌉
   if (A[mid]>k) then right ← mid
   else left ← mid
if (A[left] = k) then return True;
else return False;
```

This loop makes $\lceil \log_2(n) \rceil$ name comparisons

This function Makes $\lceil \log_2(n) \rceil + 1$ comparisons

Solution 2 – Binary search

```
Algorithm listIntersection(A,m, B,n)
Input: same as before
Output: same as before
inter ← 0
B ← sort (B,n)
for i ← 0 to m-1 do
  if (binarySearch(B, n, A[i])) then
    inter ← inter+1
return inter
```

B ← sort (B,n): $\lceil n \log_2(n) \rceil$ comparisons

for loop: $m*(\lceil \log_2(n) \rceil + 1)$ comparisons

Total number of comparisons:

$$n \lceil \log_2(n) \rceil + m*(\lceil \log_2(n) \rceil + 1) = (n + m) * \lceil \log_2(n) \rceil + m$$

Does it matter?

	Nested loops	Sort + Binary search
(m,n)	m*n	(n+m) $\lceil \log_2(n) \rceil$ + m
(8,8)	64	56
(16,16)	256	144
(32,32)	1024	352
(64,64)	4096	1086
…		
(1024,1024)	1 048 576	21 504
(10^6, 10^6)	~10^{12}	~4 * 10^7

25000 times faster!

Solution 3: Sorting and parallel pointers

```
Algorithm ListIntersection (A,m, B,n)
Input: Same as before
Output: Same as before
inter ← 0
A ← sort (A,m)
B ← sort (B,n)
PtrA ← 0
PtrB ← 0
while ( ptrA < m and ptrB < n) do {
     if ( A[ptrA] = B[ptrB] ) then {
          inter ← inter+1
          ptrA ← ptrA +1
          ptrB ← ptrB +1
     }
     else if ( A[ptrA] < B[ptrB] ) ptrA ← ptrA+1
          else ptrB ← ptrB+1
}
return inter
```

Worst case: the two lists are disjoint: (m+n) *2 comps

Total: $m \lceil \log_2(m) \rceil + n \lceil \log_2(n) \rceil + 2*(m+n)$

Solution 4: Merge-then-sort

```
Algorithm ListIntersection (A,m, B,n)
Input: Same as before
Output: Same as before
inter ← 0
Array C[m+n];
for i ← 0 to m-1 do C[i] ← A[i];
for i ← 0 to n-1 do C[ i+m ] ← B[i];
C ← sort( C, m+n );
ptr ← 0
while ( ptr < m+n-1 ) do {
    if ( C[ptr] = C[ptr+1] ) then {
         inter ← inter+1
         ptr ← ptr+2
    }
    else ptr ← ptr+1
}
return inter
```

sort: $(m+n) \lceil \log(m+n) \rceil$

Worst case: The lists are disjoint: (m+n-1) comps.

Total: $(m+n) * (\lceil \log(m+n) \rceil) + m + n - 1$

Summary

- Algorithms can be described at different levels. Pseudo-code is appropriate for human
- Many algorithms exist for solving any problem.
- For big inputs, good algorithms and data structures make a BIG difference