Data Structures and Algorithms: Lecture 1

Barbara Morawska

July 24, 2018

Algorithm

Algorithm

- well-defined computational procedure
- transforms input into output
- a tool to solve a well-specified computational problem

Computational problem is given by:

Input: ...
Output: ...

Computational problem

Computational problem is given by:

Input: ...

Output: ...

Example: sorting problem

Input: $< a_1, a_2, \ldots, a_n >$ (a sequence of n numbers)

Output: $< a'_1, a'_2, \ldots, a'_n >$ (a permutation of $< a_1, a_2, \ldots, a_n >$), such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Example: instance of a problem

Input: $< 31, 41, 59, 26, 41, 58 >$ (a sequence of 6 numbers)

Output: $< 26, 31, 41, 41, 58, 59 >$

Examples of computational problems

Example: shortest path problem

Input: A graph G and two vertices: A and B.

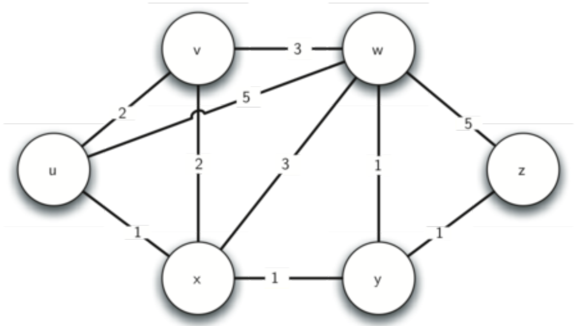
Output: A shortest path between A and B in G.

Example: traveling salesman problem

Input: A graph G and a set of vertices: $\{A_1, \ldots, A_n\}$.

Output: A sequence of vertices $< A'_1, \ldots, A'_n >$ such that the path going through all the vertices and back to the first one is of the minimal length.

shortest path and traveling salesman problems



Correctness

An algorithm is correct...

- for any input instance
- it halts
- with correct output

We say that it solves the computational problem.

An algorithm is incorrect if

- for some instances
- it does not halt
- or it halts with an incorrect answer.

Efficiency

Comparison of running times?
Insertion sort: $c_1 \times n^2$
Merge sort: $c_2 \times nlgn$

Sorting problem

Input: $< a_1, a_2, \ldots, a_n >$ (a sequence of n numbers)
Output: $< a'_1, a'_2, \ldots, a'_n >$ (a permutation of $< a_1, a_2, \ldots, a_n >$),
such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Insertion Sort:

- Keep unsorted cards on the table
- Keep sorted cards in the left hand
- Insert card from the table into the correct position in your left hand. (Comparing from right to left.)

Insertion Sort

Procedure Insertion sort(A – an array of numbers)

Input : A (an array of unsorted numbers)

Output: A the same array with numbers in nondecreasing order

for $j = 2$ *to* $A.length$ **do**
    $key = A[j]$
    `// Insert` $A[j]$ `into` $A[1..j-1]$
    $i = j - 1$
    while $i > 0$ *and* $A[i] > key$ **do**
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

Example

$A =< 5, 2, 4, 6, 1, 3 >$

Pseudocode conventions

- No "begin – end" statements. Replaced by indentation.
- "//" starts a comment
- Arrays. Array elements: $A[i]$ – i'th element of an array A
- two dots indicate range of an array:
  $A[1..j] =< A[1], A[2], \ldots, A[j] >$
- Objects have attributes: array has attribute *length*, e.g. $A.length$.
- variable is treated as a pointer:
- $y = x$ (y is assigned the pointer to x), implies $x.f = y.f$,
- $x.f = 3$ ($x.f$ is assigned value 3) implies $y.f = 3$.
  $x.f.g$ $((x.f).g)$
- *NIL* – a pointer referring to nothing
- more......

Model of computer for measuring computation time

RAM (random-access machine)

- one processor
- instructions executed one after the other

Basic instructions: take constant time each

- arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling
- data maintenance: load, store, copy
- control: conditional, unconditional, branch, subroutine call, return

Data types of RAM

integer, floating point (real numbers)

Ignored memory type and size

cache, virtual memory, etc.

Analysis of running time

Time of computation depends on size of input.

What is a size of input?

A number of items in the input.

An item is a part of input on which we do some operations.

e.g. integers, bits of a binary representation of a number, vertices in a graph, edges

Running time

The number of primitive operations (steps). Machine independent. Each step needs constant time to execute.

e.g. calling a subroutine requires constant time. (Execution of this subroutine requires various times.)

Analysis of Insertion sort

Assign some constant time cost for each step:

line	instruction	cost	nbr of times
1.	for $j = 2$ to $A.length$	c_1	n
2.	$key = A[j]$	c_2	$n-1$
3.	comment		
4.	$i = j - 1$	c_4	$n-1$
5.	while $i > 0$ and $A[i] > key$	c_5	$\Sigma_{j=2}^{n} t_j$
6.	$A[i+1] = A[i]$	c_6	$\Sigma_{j=2}^{n}(t_j - 1)$
7.	$i = i - 1$	c_7	$\Sigma_{j=2}^{n}(t_j - 1)$
8.	$A[i+1] = key$	c_8	$n-1$

Where t_j is the number of times $A[i]$ has to be compared with key.
E.g. for $j = 5$, t_j maybe 4.

Running time of Insertion Sort $T(n)$

$$
\begin{aligned}
T(n) = {} & c_1 n + c_2(n-1) + c_4(n-1) + c_5\Sigma_{j=2}^{n} t_j \\
& + c_6\Sigma_{j=2}^{n}(t_j - 1) + c_7\Sigma_{j=2}^{n}(t_j - 1) + c_8(n-1)
\end{aligned}
$$

Best case: array already sorted in increasing order

- while-loop not entered
- no shifting (line 6,7)
- $t_j = 1$ for $j = 2, 3, \ldots, n$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) \\
&\quad + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

$T(n) = an + b$ (linear function of n)

Worst case: array ordered in decreasing order

- $t_j = j$, for $j = 2, 3, \ldots, n$
- line 5: $\Sigma_{j=2}^{n} t_j = \Sigma_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$
- line 6, 7: $\Sigma_{j=2}^{n} t_j - 1 = \Sigma_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$

$$\begin{aligned}
T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) \\
+ c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2}) + c_8(n-1) \\
= (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + \\
(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_6)n \\
- (c_2 + c_4 + c_5 + c_8)
\end{aligned}$$

$T(n) = an^2 + bn + c$ (quadratic function of n)

Note on average case

Average case for Insertion sort

For each j, $j = 2, 3, \ldots, n$, half of the numbers in $A[1..j-1]$ and smaller than j and half are bigger.

$t_j = j/2$

In the average case $T(n)$ is a quadratic function in n. (The same as in the worst case.)

We will focus on the worst case, which is the upper bound on the running time for any input.

Two ways of designing algorithms

- incremental approach (insertion sort)
  incrementing solved part step by step
- "divide and conquer"
  - divide the problem into smaller ones
  - conquer the subproblems by recursive calls of the procedure
  - combine the solutions to the subproblems to the solution of the main problem.

  Example: Merge sort.

Merge sort

- divide: divide the sequence of n-elements into two subsequences of $n/2$ elements each
- conquer: call *Merge sort* recursively on small subsequences
- combine: *merge* two sorted subsequences to get the sorted sequence.

Merge

Merge(A, p, q, r)

- Input: two arrays (parts of one array A) $A[p..q]$ and $A[q+1..r]$,
- in each elements are in increasing order
- $p \leq q < r$
- Ouput: a merged array $A[p..r]$ of elements in increasing order
- $A[p..r]$ has $r - p + 1$ elements

Merge

Procedure Merge(A, p, q, r)

Input : A, p, q, r where $A[p..q], A[q+1..r]$ are arrays of numbers sorted in increasing order, $p \leq q < r$

Output: $A[p..r]$ an array with numbers in increasing order

$n_1 = q - p + 1$ `// length of` $A[p..q]$
$n_2 = r - q$ `// length of` $A[q+1..r]$
Let $L[1..n_1+1]$ be a new empty array. `// One additional place for a` *sentinel*
Let $R[1..n_2+1]$ be a new empty array. `// One additional place for a` *sentinel*
`// Copy parts of the input array into two new arrays`
for $i = 1$ *to* n_1 **do**
    $L[i] = A[p+i-1]$
for $j = 1$ *to* n_2 **do**
    $R[j] = A[q+j]$
$L[n_1+1] = \infty$ `//` *sentinel*
$R[n_2+1] = \infty$ `//` *sentinel*

```
i = 1
j = 1
// Redefine A[p..r] here
for k = p to r do
    if L[i] ≤ R[j] then
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
```

Question: what is the role of sentinels?

Running time of Merge

- Steps 1 – 4: constant time
- 5 – comment
- Steps 6 – 9 (copying to two arrays): $\Theta(n_1 + n_2) = \Theta(n)$
- Steps 10 and 11 (sentinels): constant time
- 14 – comment
- Steps 15 to 21 (redefining array $A[p..r]$): $r - p + 1 = n$ times constant time required for comparison. Hence $\Theta(n)$.

Hence Merge requires $\Theta(n)$ time.

Merge sort

Procedure Merge-sort(A, p, r)

Input : A, p, r (array and two indexes in this array $p \leq r$)
Output: $A[p..r]$ an array with numbers in increasing order

if $p < r$ `// if` $p = r$ `then` $A[p..r]$ `contains one element`
 then
    $q = \lfloor \frac{p+r}{2} \rfloor$
    `Merge-sort`(A, p, q)
    `Merge-sort`$(A, q + 1, r)$
    `Merge` (A, p, q, r)

Initially we call `Merge-sort(A, 1, A.length)`.

Analyzing "divide-and-conquer" algorithms

Create a recurrence equation

Let $T(n)$ be the running time of the algorithm depending on n

- (base case) small n, $n \leq c$ for constant c then the algorithm takes constant time $\Theta(1)$
- division yields a (a number of) smaller subproblems each with input size n/b;

Time:

- $D(n)$ - for divide stage
- $T(n/b)$ - for computing solution for each subproblem
- $C(n)$ - for combining solutions for subproblems

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{if } n > c \end{cases}$$

Analysis of `Merge-sort`

assume that n is a power of 2 (division by 2 yields always an integer)

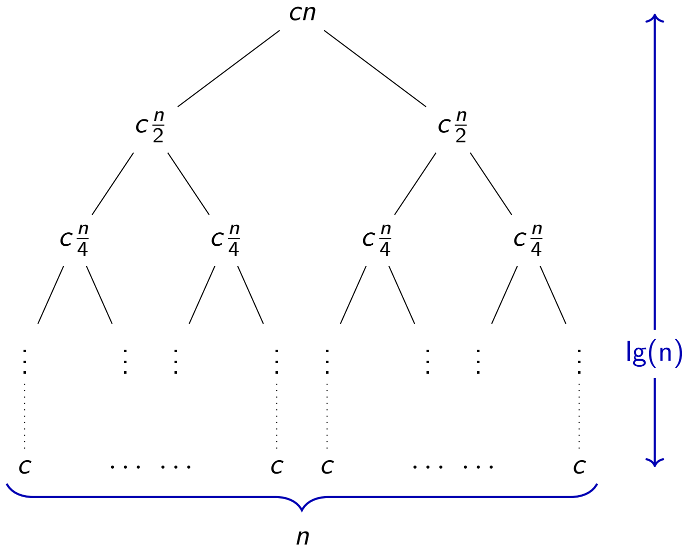
- (base case) if $n = 1$, `Merge-sort` takes constant time: $T(1) = \Theta(1)$
- Divide: compute the middle $n/1$ takes constant time: $T(1) = \Theta(1)$
- Conquer: $2T(n/2)$
- Combine: `Merge` takes linear time: $C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2\,T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

$$= \Theta(n\,lgn)$$

Recursion tree for a recurrence

$$T(n) = 2\,T(n/2) + cn$$



Number of levels in a recursion tree

Theorem

Total number of levels of the recursion tree with n leaves is $\lg(n) + 1$.

Note: the level of the root is counted as 1 not 0.

Proof.

Proof is by induction on the number of leaves n.
Base case: $n = 1$ If the tree has 1 leaf only, $\lg(1) + 1 = 0 + 1 = 1$.
Induction hypothesis: For tree with $n = 2^i$ leaves, the number of levels is $\lg(2^i) + 1 = i + 1$.
Induction step: prove the statement for $n = 2^{i+1}$ (assumed n is a power of 2). The number of leaves is $2^{i+1} = 2^i \times 2$. Hence the tree has two subtrees with 2^i leaves and are connected at the top by one level. By induction hypothesis each of the subtrees has $\lg(2^i) + 1 = i + 1$ levels. Hence the main tree has $i + 1 + 1 = \lg(2^{i+1}) + 1$ levels. $\square$