Algorithms and Data Structures

Saul Pierotti

March 23, 2020

Introduction

- Exam will be written, but you can do oral if you want
  - After lectures, around June 25
- An algorithm is a finite series of steps that solves a problem
  - In CS, an algorithm is a well defined computational procedure
- For a sorting algorithm, the input is a series of numbers and the output is an ordered series of numbers
- Algorithms are for humans, while a program is for a computer
- Algorithms are written in pseudocode, which follow specific conventions
- A problem can be solved by many algorithms
- An algorithm can be implemented in many different programs
- Properties of algorithms
  - Input for an algorithm can have 0 or more inputs
  - It always as 1 or more outputs
  - It should be clearly defined and unanbiguous
  - It should terminate after a finite number of steps
  - All operations must be basic
    * They can be solved exactly and in finite time
- The correctness of an algorithm is difficult to prove
  - I would need to try all possible inputs (!)
  - Published algorithms have a mathematical proof
- Incorrect algorithms can produce a wrong output or not produce any for some instances
  - In some cases they are still useful, if I can control their error rate
- Efficiency is related to the ability of an algorithm to be executed with available resources
- Resources are time and memory
- Time is measured in running time, not CPU time
  - CPU time is dependent on CPU (!)
  - CPU time is number of instruction divided by number of istructions per unit time
  - Running time is the number of primitive operations to be performed in proportion to the input size
- The algorithm influences time much more than hardware, we do not focus on hardware (!)
- Running time of n^2 is unacceptable for large inputs
- Using the right data structure is important for efficiency
- Decision trees are essential in CS
- An instace of a problem is a specific input for that problem
- In a while and for loop, the test is always executed once more than the body

Math backgroud

- Finite sums
  - $\sum_{k=1}^{n} a_k = a_1 + a_2 + ... + a_n$
- Infinite sums

    - $\sum_{k=1}^{\infty} a_k = a_1 + a_2 + ...$
- Sums are linear
    - $\sum_{k=1}^{n}(ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k$
- The arithmetic series
    - $\sum_{k=1}^{n} k = 1 + 2 + ... + n = \frac{n(n+1)}{2}$
- The quadratic arithmentic series
    - $\sum_{k=1}^{n} k^2 = 1 + 4 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$
- The cubic arithmetic series
    - $\sum_{k=1}^{n} k^3 = 1 + 8 + ... + n^3 = \frac{n^2(n+1)^2}{4}$
- The geometric series
    - $\sum_{k=1}^{n} x^k = 1 + x + x^2 ... + x^n = \frac{x^{n+1}-1}{x-1}$
    - When x is less than 1
        * $\sum_{k=1}^{\infty} x^k = \frac{1}{1-x}$
        * $\sum_{k=1}^{\infty} kx^k = \frac{x}{(1-x)^2}$
- Other formulas
    - $\sum_{k=1}^{n} \log k \approx n \log n$
    - $\sum_{k=1}^{n} k^p \approx \frac{n^{p+1}}{p+1}$
- A set is a non-ordered and non repetitive collection of elements
- Sets can be infinite
- It is described as $S = ...$
- 2 sets are equal if they contain the same elements
- The cardinality of a set $|S|$ is the number of elements it contains
- If A contains all the elements contained in B and also other elements, then B is a proper subset of A
- The power set $P(S)$ is the set of all subset of S, including the empty set and S itself
    - $|P(S)| = 2^{|S|}$
- The cartesian product of 2 sets is a set containing all possible pairs of elements
- The cartesian product of n sets is a set of n-tuples
- A tree has only one way to go from one node to the other
- A graph can have cycles
- A forest is made of many trees
- Any non-empty tree with n nodes has n-1 edges
    - If this is not true, we don't have a tree
- A tree is rooted if one of its nodes is distinguished as root
    - It can be defined recursively such that every non-root node of a rooted tree is itself the root of a subtree
- Tree terminology is similar to that of ancestry trees
- The depth of a node is its distance from the root (number of edges)
- The height of a node is the lenght of the longest path to a leaf
- A binary tree is an order tree with 2 subtrees wich are themselves binary

Pseudocode

- We start counters from 1 since it is easier to understand
- Bold words are reserved words like **return**
- Variables are always local to the current procedure
- We can have loops like **while** and **for**

```
for i=0 to/downto i=4 (by 3)
    <statement>
```

- We have if statements

```
 if <condition>
    <statement>
```

- Comments are rendered with //
- No colon/semicolon at the end of lines
- Use of indented blocks
- Differentiate conditional expressions and assignments (!)
- A slice of an array is indicated as `A[3..5]`
- Attributes of objects are indicated as `object.attribute`
  - The lenght of an array can be indicated as `A.lenght`

Sorting

- Sorting is an intermediate step in many tasks in CS
- There are many sorting algorithms

Insertion sort

- It is like arranging card in order in your hand by picking one at a time
- I take 1 unsorted object at a time and I insert it in the correct position in the sorted array
  - I compare with all the objects in the sorted array, until I find the right position
- I start from the first element of the array and I don't do anything
- I take the second element, and if it is smaller than the first I swap them
- I take the third, and if it is smaller than the second I compare it with the first and I swap in the right position
- I continue like this for all the elements

Pseudocode

```
INSERTION-SORT(A)
    for j = 2 to A.lenght
        key = A[j]
        i = j-1
        while i > 0 and A[i] > key
            A[i+1] = A[i]
        A[i+1] = key
```

Running time

- Nearly sorted numbers can be sorted much fatser with insertion sort
- The input size is the length of the array
  - `n = A.lenght`
- The initial **for** test is executed n times
  - It is n, not n-1 because even when it is false we still have to check ones (!)
  - The body of the **for** is executed n-1 times
- The assignemt of key is therefore executed n-1 times
- The **while** test is executed $\sum_{j=2}^{n} t_j$ times
  - The body of the **while** is executed $\sum_{j=2}^{n} t_j - 1$
  - There are 2 assignments on the while body
- The final assignment after the **while** inside the **for** is executed n-1 times

Best case

- The array is already sorted
- I never enter the while, but I do completely the for
- This means that t_j is 1, I only do the test
- The time is linear

Worst case

- The array is in reverse sorted order
- The time is quadratic

Average case

- It is really difficult to do, we prefer to focus on the worst case

Evaluation

- For almost sorted sequences its running time is almost linear
- Can be online
  - It can sort sequences as they arrive
- In the worst and average cases it is quadratic
  - Quadratic is really bad (!)

Merge sort

- It is a divide and conquer algorithm
  - Divide a problem in subproblems of smaller size
  - Solve the subproblems recursivley (conquer)
  - Combine the solution to solve the original problem
- The complicated part is the merging process
- It runs always as n*log(n), there is not worst or best case
- It requires a lot of memory to store all the sub-arrays
- It is worse than insertion sort in the best case, but better in most cases
- It cannot work online (!)

Idea

- I want to sort the array A
- I split the array using the indeces p,q,r such that $p <= q < r$
- I want to produce a single sorted subarray
- I call initially on A with p=1 and r=A.lenght
- The index q is the one that best splits the array in 2
- For merging I always have sorted arrays to merge
  - The first element of each array is guaranteed to be the smallest one of the entire array
  - I compare the first element of the 2 arrays to be merged, and I put the smallest in the output array
  - I repeat until one of the arrays is empty
  - I finsih by putting what remains of the other array in the output
  - I put an immaginary infinite at the end of any array
    - This is so that when I finish the elements of an array, whatever remains in the other is smaller and so it is inserted in the output

Pseudocode

```
MERGESORT(A,p,r)
    if p < r
        q = (p+r)/2
        MERGESORT(A,p,q)
        MERGESORT(A,q+1,r)
        MERGE(A,p,q,r)

MERGE(A,p,q,r)
    n1 = q - p + 1
    n2 = r - q
    for i = 1 to n1
        L[i] = A[p+i-1]
```

```
    for j = 1 to n2
        R[i] = A[q+j]
    L[n1+1] = \infty
    R[n2+1] = \infty
    i = 1
    j = 1
    for k = p to r
        if L[i] <= R[j]
            A[k] = L[i]
            i += 1
        else A[k] = R[j]
            j += 1
```

`MERGESORT(A,1,A.lenght)`

Running time

- MERGE
  - Copying the elements into the subarrays takes $\Theta(n)$
  - Adding elements to the final array takes n iterations of that themselves take constant time
    * $\Theta(n)$
  - In total, the merging takes $\Theta(n)$
- MERGE-SORT
  - Let T(n) be the unknown running time of MERGE-SORT
  - Calculating q: $\Theta(1)$
  - Solve recursively 2 subproblems of size n/2: 2T(n/2)
  - Call to MERGE: $\Theta(n)$
  - So, $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(1) + \Theta(n) & \text{if } n > 1 \end{cases}$
  - The recursive equation can be solved and we find that $T(n) = \Theta(n \log n)$

Limiting behaviour of functions

- There are different notations to define the behaviour of functions
- The Θ (theta) notation signifies asymptotic equality
  - Formally, for a function $f(n)$ having a certain Θ notation there are 2 constant that multiplied for the Θ function are constantly greater or smaller than $f(n)$ for $n > n_0$
    * This is defined as a tight bound
  - If I say that $f(n) = \Theta(g(n))$ I mean that $f(n)$ belongs to the family of functions with order of growth $g(n)$
- The O (big-O) notation indicates an upper bound for the asymptotic behaviour
  - The formal definiton is similar to that of Θ, but instead of a tight bound I only search for an upper bound
    * I only want a constant, not 2 (!)
- The Ω (big-Omega) notation indicates a lower bound for the function
- An important theorem: $f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$
- Factorials are faster than exponentials, but slower than n^n (!)

Designing algorithms

- A recurrence equation describes a function in terms of its value on a smaller input
- An example: analysis of a divide and conquer algorithm

- $T(n)$ is the running time of the algorithm on input n
  - Dividing takes $D(n) = \Theta(1)$ time
  - Conquer takes the same T on a smaller input, so $aT(n/b)$
    - I need to solve a subproblems in with an inpuit size reduced of a factor b
  - Combining the solutions takes $C(n) = \Theta(n)$ time
  - So we have that $T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + c + cn & n > 1 \end{cases}$
- Solving recurrence equations: the iteration method
  - If I have $T(n) = T(n/2) + c \implies T(n/2) = T(n/4) + c$ and so on
  - This implies that $T(n) = c + c + T(n/4)$
  - If I continue this until I get to the base case $T(1)$
  - I can write therefore $T(n) = c * k + T(n/2^k)$
  - The base case will be when $n = 2^k$ and therefore $k = \log n$
  - So I get that $T(n) = c * \log n + T(n/2^{\log n}) = c * \log n + T(1)$
  - This means that $T(n) = \Theta(\log n)$
- Solving recurrence equations: the recursion tree method
  - I convert the equation into a tree and sum up the cost of each node
  - Let's try with $T(n) = 2T(n/2) + n^2$
  - The root has a cost n^2 and 2 children of cost $T(n/2)$ each
  - I continue to expand until the base case
  - This gives us a $O(n^2)$, since I cannot determine a tight bound

Heapsort

- Normally, the numbers to be sorted are a key that is paired to other data, forming a record
  - A record is composed of a key and satellite data
- When I want to sort, are all keys unique?
- Its running time is $\Theta n \log n$
- It is based on a data structure called heap
  - An heap is a nearly complete binary tree
    - All nodes are binary except for possibly the last level
    - If the last level is not full, it is filled from left to right
  - It follows the heap property: the value of a parent must be greter that that of a children
    - This is a max heap, there are also min heaps where the property is opposite
  - The size of an heap is the number of nodes
- We can represent an heap with an array
  - The first element is the root
  - The children of the root are the second and third element
  - The fourth and fifth element are the children of the second, and so on
  - The children of node A[i] are nodes A[2i] and A[2i+1]
  - The parent of A[i] is $A[\lfloor i/2 \rfloor]$
  - The maximum element is always the root
- How to maintain the max heap property (MAX-HEAPIFY)
  - I recursively explore the tree
  - If I find a parent smaller than its child, I swap them and continue
  - I assume that there is only one violation
  - The running time is O(log n), or linear to heap size (O(h))

```
MAX-HEAPIFY(A,i)
    l, r = 2i, 2i+1
    if l <= A.heap-size and A[l]>A[i]
        largest = l
    else largest = i
```

```
    if r <= A.heap-size and A[r]>A[i]
        largest = r
    else largest = i
    if largest != i
        exchange A[i] and A[largest]
        MAX-HEAPIFY(A,largest)
```

- Now we start from a random array and we want to make it a max heap
  - Note that $A[(\lfloor n/2 \rfloor + 1)...n]$ are leaves

```
BUIL-MAX-HEAP(A)
    A.heap-size = A.lenght
    for i = floor(A.lenght/2) downto 1
        MAX-HEAPIFY(A,i)
```

- This operations has an loose upper boundf of O(n log n)
  - I do n/2 times a O(log n) operation
- However, the argument to MAX-HEAPIFY is almost never n (!)
  - In the first step it is 1, then 2 and so on
- The worst case running time of MAX-HEAPIFY is O(log i) where i is the value of the for loop in BUILD-MAX-HEAP
  - We obtain O(n)
- The next step is to actually sort the array
  - We swap the root with the last element and decrease the heap-size by 1
  - We call MAX-HEAPIFY on the root to rebuil the max-heap property
  - We repeat until the heap-size is 1
  - The array is sorted (!)

```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i = A.lenght down to 1
        exchange A[1] with A[-1]
        A.heap-size = A.heap-size - 1
        MAX-HEAPIFY(A,1)
```

- The total running time is O(n log n)