Warm Up: Analysis of Some Basic Algorithms

1 Finding maximum

This is an easy problem to formulate and solve. We are given n numbers $X = x_1, \ldots, x_n$, and we are asked to find a maximum of these numbers. These n numbers are presented as an array or a linked list. The algorithm that finds the maximum reads the integers in the list one by one, keeping the current estimate of the maximum value, as it moves on. These type of algorithms are sometimes called "one-pass style" algorithms.

Algorithm 1 FindMax (X):

$max = x_1$
For $i = 2$ to n
$-$If $x_i > max$ then set $max = x_i$
Endfor

In this algorithm, the statements "If $x_i > max$ then set $max = x_i$" are executed each in constant time. In other words, the algorithm does a constant work for each element of the list. Therefore, on input size n, the running time of the algorithm is linear on the size of the input.

This sort of one pass algorithms appear in practice quite often. For instance, inputs can be data streams such as computer network traffic, phone conversations, ATM transactions, twitter messages, stock-exchange information. Often, you need to design applications that process such data streams. Such applications are typically one-pass style algorithms.

Exercise 1. Write down an algorithm that given input t_1, ..., t_n of texts (written in english alphabet), outputs those texts t that contain any of the following words *exchange*, *escape*, *data*, and *stream*. What do you think the size of the input is in this example?

2 Merging sorted lists

This is an interesting example of an algorithm that extends the idea of "one-pass" algorithm. The algorithm runs in a linear time for reasons more subtle than "one-pass" algorithms.

Suppose we are given two *sorted* lists $A = a_1, \ldots, a_n$ and $B = b_1, \ldots, b_m$ of numbers. Both lists are assumed to be sorted in ascending order. Recall that a list is *sorted* (in ascending order) if each item in the list is greater or equal to the previous item in the list. We would like to merge these two lists into one sorted list C, also in ascending order. For example, merging the following two lists $A = 2, 2, 5, 6, 6, 9, 10, 11, 11, 11$ and $B = 1, 1, 2, 3, 4, 4, 8, 12, 12, 14$ produces the list $C = 1, 1, 2, 2, 2, 3, 4, 4, 5, 6, 6, 8, 9, 10, 11, 11, 11, 12, 12, 14$.

Obviously, the method of writing down all elements of A first and then all elements of B does not work since the resulting list might not be sorted. The idea is to start with comparing the first two elements of A and B, and writing down the one that is smallest; after that one of the lists A or B becomes smaller, and we iterate the process just described. Here is the algorithm:

Algorithm 2 Merge(A,B):

Initialise two pointers which point the front elements of A and B, respectively
While both lists are nonempty:
− Let a and b be the elements pointed by the two pointers in A and B, respectively
− If $a \leq b$ then append a to the output list; otherwise append b
− Advance the pointer in the list from which the smaller element was appended
EndWhile
Once one list is empty, append the remainder of the other list to the output.

Now we would like to analyse the running time of the algorithm. The space needed to store the input is $n + m$. Hence, $n + m$ is the size of the input. Note that the reasoning as in the algorithm above, that we do a constant work per element, does not work! The amount of time for processing each element of the input is bounded by $\max\{n, m\}$. So, the crude upper bound on running time is thus $\max\{n^2, m^2\}$, and this is not a linear time on the size of the input.

Note that every iteration takes a constant amount of time. So, we need to bound the number of iterations. Every iteration causes one of the pointers to move to the next item. Hence, the pointer for list A moves at most n times, and the pointer for B moves at most m times. Therefore, the number of iterations is bounded by $n + m$. The last instruction of the algorithm runs at most $\max\{n, m\}$ times. Hence, the running time of the algorithm is proportional to $n + m$. In other words, the algorithm runs in linear time on the size of the input.

Exercise 2. Explain why Merge(A, B)-algorithm produces a sorted array. For simplicity you can assume that all elements in A and B are pairwrise distinct.

3 Mergesort algorithm

Let $A = a_0, a_1, \ldots, a_{n-1}$ be a list of n integers. Sometimes each a_i is called an *item* of the list. We want to solve the following problem, known as *the sorting problem*:

Given a list $A = a_0, a_1, \ldots, a_{n-1}$ of n integers, sort the list and output it.

Formally, we would like to output a list $B = b_0, b_1, \ldots, b_{n-1}$ such that

1. The array $B = b_0, b_1, \ldots, b_{n-1}$ is sorted.
2. Each item of B appears in the original list A *exactly* as many times as it appears in B.
3. Each item in A is also an item in B

Let X be a list $x_0, x_1, \ldots, x_n$. The *midpoint* of X is $x_{n/2}$ if n is even; If n is odd, then the mid point is $x_{(n+1)/2}$. For example, the mid point of $(1, 4, 3, 25, 8, 9)$ is 25.

The idea of MergeSort(X) algorithm is the following. Divide X into two lists: $Left$ and $Right$, where $Left$ is the list of elements of X from the first element to the midpoint element, and $Right$ is the list of elements of X from the midpoint element to the last element in X. Now one needs to sort $Left$ and $Right$ and then merge the sorted lists using Merge(A,B)-algorithm. Here is a high level description of the MergeSort(X)-algorithm:

Algorithm 3 MergeSort(X):

If X is the empty list or contains one item only then output X.
– While X contains more than 1 item
–– Find the midpoint x_k of X
–– Define two lists $A = x_0, x_1, \ldots, x_k$ and $B = x_{k+1}, \ldots, x_{n-1}$
–– Set A=MergeSort(A) and B=MergeSort(B)
– EndWhile
EndIf
Output Merge(A,B).

This is clearly a recursive algorithm. Namely, when MergeSort(X) is in the while loop, the algorithm is applied to lists of strictly smaller length than that of X. Note that the size of the input for this algorithm is the length n of the input array.

Now we would like to bound the running time of the algorithm. So, now assume that $T(n)$ is the worst possible running time of the algorithm on all inputs X of size n. The time $T(n)$ can be bounded by the sum of the running times of the following processes:

1. Finding the mid point x_k of the list X.
2. Creating the lists A and B.
3. Running MergeSort(A) and MergeSort(B).
4. Merging sorted lists A and B returned by MergeSort(A) and MergeSort(B).

Finding the mid point x_k, if the list is implemented as an array, takes a constant time, say c_1. Creating the lists A and B takes time proportional to n. Since the sizes of A and B is now bounded by $n/2$, the running times of MergeSort(A) and MergeSort(B) both are bounded by $T(n/2)$. The time taken for merging sorted lists A and B returned by MergeSort(A) and MergeSort(B) is again proportional to n (as noted in the previous section). So, we can write the following bound:

$$T(n) \leq c_1 + c_2 \cdot n + 2 \cdot T(n/2) + c_3 \cdot n \leq Cn + 2T(n/2),$$

where $C = \max\{c_1, c_2, c_3\}$. We now can *unroll* this further:

$$T(n) \leq Cn + 2T(n/2) \leq Cn + 2(C(n/2) + 2T(n/4)) = 2Cn + 4T(n/4) \leq$$
$$2Cn + 4(C(n/4) + 2T(n/8)) = 3Cn + 8T(n/8) \leq 4Cn + 16T(n/16).$$

To simplify our reasoning, assume that n is a power of 2. Then the length of the sequence

$$n,\ n/2,\ n/4,\ n/8,\ \ldots,\ n/2^k,\ \ldots,\ 2,\ 1$$

equals $log(n)$. Also, from the properties of the log function we know that $2^{log(n)} = n$. So, continuing on the process above we arrive at the following:

$$T(n) \leq 4Cn + 16T(n/16) \leq \ldots \leq Cn \cdot log(n) + 2^{log(n)}T(1) \leq C'n \cdot log(n).$$

Thus, this analytical reasoning shows that the running time of the MegeSort(A)-algorithm is bounded by a function proportional to $n \cdot log(n)$. It is possible to show that this is the best bound, but this would be out of scope of this course.

Below we present a pseudocode implementation of MergeSort(X)-algorithm:

Algorithm 4 Implementation of MergeSort(X):

```
INPUT (unsorted) array X = x_0, x_1, ..., x_{n-1}
OUTPUT (sorted) array X
  Create a new array Y of length n
  MergeSortRec(X, Y, 0, n − 1)
  return X

SUBROUTINE MergeSortRec(data, temp, low, high)
INPUT length-n arrays data & temp, integers low&high
  Set n ← high − low + 1
  Set mid ← low + n/2
  if n < 2 then return end if
  for i = low, ..., mid − 1 do
      temp[i] ← data[i]
  end for
  MergeSortRec(temp, data, low, mid − 1)
  MergeSortRec(data, temp, mid, high)
  Merge(data, temp, low, mid, high)

SUBROUTINE Merge(data, temp, low, mid, high)
INPUT length-n arrays data & temp, integers low, mid, high
  set r ← 0                                   ▷ the starting index in result
  set t ← low                                 ▷ the starting index in temp
  set d ← mid                                 ▷ the starting index in data
  Create a new array result of length high − low + 1
  while t < mid & d ≤ high do
      if data[d] < temp[t] then
          result[r] ← data[d]
          r ← r + 1; d ← d + 1
      else
          result[r] ← temp[t]
          r ← r + 1; t ← t + 1
      end if
  end while
  while t < mid do
      result[r] ← temp[t]
      r ← r + 1; t ← t + 1
  end while
  while d ≤ high do
      result[r] ← data[d]
      r ← r + 1; d ← d + 1
  end while
  for i = 0, ..., high − 1 do
      data[low + i] ← result[i]
  end for
```

4 Closest pair problem

This is a geometric problem that has many applications in computer vision. We are given n points $P = p_1, \ldots, p_n$ in the plane. Each point p_i is given by coordinates (x_i, y_i). We need to find a pair of points closest to each other. There is a natural brute-force algorithm for this problem. Enumerate all pairs of points, compute the distance between each pair, and choose the pair with the smallest distance. The size of the input P is n. The algorithm makes n^2 iteration. Each iteration takes a constant time. Hence, the running time is bounded by n^2.

Algorithm 5 ClosestPair (P):

For each input point p_i
– For each other input point p_j
– Compute distance $d(p_i, p_j)$ between the points
– If $d(p_i, p_j)$ is less than the current minimum, update minimum to $d(p_i, p_j)$
– Endfor
Endfor

Quadratic time arises when one needs to enumerate all pairs of items in the input, and spend a constant time on processing each such pair. In terms of implementation, this tells us that there are two nested loops: "outer loop" and "inner loop". For instance, in the algorithm above, the "inner loop" over the points p_j has n iterations, and the "outer loop" over points p_i also has n iterations, where each iteration of the outer loop invokes n iterations of the "inner loop".

5 Cubic time algorithms

Cubic time algorithms arise when they use three nested loops. Assume that the input for a such algorithm has size n. Imagine that each iteration of the first loop invokes n iterations of the second loop, and each iteration of the second loop invokes n iterations of the third loop, and the third loop makes n iterations each taking a constant time. Intuitively, this produces a running time proportional to n^3.

One example is this. Suppose we have n sets

$$S = S_1, S_2, \ldots, S_n,$$

where each S_i is a subset of $\{0, 1, \ldots, n\}$. We want to know whether there are two disjoint sets; in other words, we want to know if there are two sets S_i and S_j such that they have no elements in common. The size of this problem is defined to be n. This definition of size in this problem is reasonable as we have postulated that the sets $S_1, \ldots, S_n$ are subsets of $\{0, 1, \ldots, n\}$. Without this assumption, a better definition of size would be more explicit definition: the sum of the cardinalities of the sets $S_1, \ldots, S_n$. Recall that the cardinality of a set is the number of its elements.

A straightforward high-level description of an algorithm that solves the problem is this:

For pair of sets S_i and S_j
– Determine whether S_i and S_j have an element in common
Endfor

For the analysis of the running time, we need to refine the algorithm above. For this we present slightly detailed algorithm with nested loops.

Algorithm 6 FindDisjoint (S):

For each set S_i
– For each other set S_j
–– For each element p in S_i
––– Determine whether p also belongs to S_j
–– Endfor
– If no element of S_i belongs to S_j then output S_i and S_j
– Endfor
Endfor

Now we can give a more routine explanation of the running time. Each sets S_i has at most n elements. Hence, the inner loop makes at most n iteration. Each iteration takes a constant time to run. Each such sequence of iterations of the inner loop is within one iteration of the loop for S_j. There are n iterations of the while loop for S_j (since j runs from 1 to n). These n iterations of the loop for S_j are inside of one iteration of the outer loop. So, multiplying these 3 factors together we bound the running time of the algorithm by a function proportional to n^3.

6 Euclidean algorithm

Euclid invented the algorithm 300 years BC. The algorithm, given positive integers n and m, produces the greatest common divisor for n and m. We explain the problem in more detail.

Given two positive integers n and m, a *common divisor of n and m* is an integer k that divides both n and m. Clearly, 1 is a common divisor of any two integers n and m. As an example, consider the numbers 18 and 24. The divisors of 18 are 1, 2, 3, 6, 9, 18. The divisors of 24 are 1, 2, 3, 4, 6, 8, 12, 24. Their common divisors are 1, 2, 3, and 6. The *greatest common divisor of n and m*, written $\gcd(n, m)$, is the largest common divisor of n and m. Thus, for the the example above, we have $gcd(18, 24) = 6$.

We can compute $gcd(n, m)$ as follows. Construct two lists; the first is the list of the divisors of n and the second is the list of the divisors of m. Find the largest number that occurs in *both* lists. This can be found, for instance, by Algorithm 2. This largest number is then $gcd(n, m)$.

What is the running time of this algorithm? To answer this question, we need to know the *size* of the input!

What is the size of an integer n? The answer depends on the way we represent n. If n is represented in unary, then representation of n is just a sequence of 1s $111\ldots1$; the length of the sequence is n. So, in unary representation the length of n coincides with the value of n. This presentation of n is naive and not compact. So, we use the presentation that we are accustomed to: the decimal representation. For instance, the integer 100000000 represents integer n with value hundred million, and the size of this integer is 9.

Now we answer the question about the running time of the algorithm above. To list all divisors of n, we need to go through all $i < n/2$ and check if i divides n. If i divides n, then i is put into the list of divisors of n. The running time of this process is bounded by $n/2$. Similarly, we can list all divisors of m in time bounded by $m/2$. To find the greatest common divisor we pass through the first list and the second list of divisors. Thus, the process of finding the greatest common divisor is proportional to $n+m$. If n and m are represented in unary then this algorithm is efficient in the sense it runs in linear time on the size of the input. However, in case we use the usual (decimal) representation, then the algorithm runs roughly in time proportional to the exponent of the input size! So, the algorithm above is *not* efficient.

Below we provide an algorithm invented by Euclid. The algorithm is simple and, mysteriously, it always finds the greatest common divisor for any input n and m. The only thing required to follow the algorithm is to know one simple fact about dividing one integer by another.

The fact is the following. Let a and b be positive integers. Then there always exist q and r such that $a = q \cdot b + r$ and $0 \leq r < b$. This division process is used at every iteration of the while loop in the algorithm.

Algorithm 7 *Euclidean*(n, m)

If $n = m$ then output n as $gcd(n, m)$ and stop
If $n > m$ then initialize $a = n$ and $b = m$; Otherwise, initialize $a = m$ and $b = n$
While $r \neq 0$
− Divide a by b by finding integers q and r such that $a = q \cdot b + r$, where $0 \leq r < b$
− Set $a = b$ and $b = r$
EndWhile
Output b.

As an example, consider the case when the input values of n and m are: $n = 2528$ and $m = 340$. We apply the Euclidean algorithm step by step as presented in the table below. The initial values of a and b are the following: $a_1 = 2528$ and $b_1 = 340$.

Iteration	a	b	Divide a by b
1	$a_1 = 2528$	$b_1 = 340$	$2528 = 7 \cdot 340 + 148$
2	$a_2 = 340$	$b_2 = 148$	$340 = 2 \cdot 148 + 44$
3	$a_3 = 148$	$b_3 = 44$	$148 = 3 \cdot 44 + 16$
4	$a_4 = 44$	$b_4 = 16$	$44 = 2 \cdot 16 + 12$
5	$a_5 = 16$	$b_5 = 12$	$16 = 1 \cdot 12 + 4$
6	$a_6 = 12$	$b_6 = 4$	$12 = 3 \cdot 4 + 0$. Here $r = 0$ Output $gcd(2528, 340) = 4$

Exercise 3. For this exercise, the sizes of n and m are the lengths of the decimal representations of n and m. Explain why Euclidean algorithm runs in time proportional to the size of the input. For the analysis, assume that the division process takes a constant amount of time.