Com S 311 Section B
Introduction to the Design and Analysis of Algorithms

Lecture Two for Week 1

Xiaoqiu (See-ow-chew) Huang

Iowa State University

January 28, 2021

Analysis of Algorithms

Input: Two arrays A and B of the same length and without duplicates.

Problem: Do A and B contain the same elements?

Which of the following strategies is better?

Strategy 1

```
for each index i in array A
  if A[i] does not appear in array B
    return false
return true
```

Strategy 2

```
make a copy of each array and sort both arrays
for each index i
  if A[i] is different from B[i]
    return false
return true
```

Some Measures of the Quality of Algorithms

The amount of time taken by an algorithm

The amount of memory ...

The amount of network bandwidth ...

The amount of effort to code an algorithm

Under each measure, a better algorithm takes a smaller amount of resource or effort.

We will focus on techniques for estimating the running time of an algorithm without turning the algorithm into a program.
However, the techniques can apply to other measures too.

Concepts in Algorithms

A *problem* consists of a large number of instances.

An *instance* of the sorting problem is a sequence of numbers: 5, 57, 23, 49, 34.

The size of an instance is the *input size*, defined as the amount of memory needed to store the instance.

An input instance consists of n numbers with each number taking 64 bytes of memory. The input size is $64 \times n$.

We only need to compute the input size within a constant factor (accurate to a constant factor).

Thus, the input size in the above example is n, ignoring the constant factor 64.

Concepts in Algorithms

What is the input size of an instance consisting of two numbers each with n digits?

The input size is $2n$, or just n, but not 2.

Running Time of an Algorithm

The running time of an algorithm on an instance is the number of primitive instructions executed.

An algorithm may take more time on some instances than on other instances of the same size.

The running time of an algorithm may be expressed as the function of the input size obtained by taking the average over all possible instances of the same size. However, an average case analysis is typically difficult.

Thus we often focus on finding the worst-case running time of an algorithm, the longest running time on any instance of size n, which is expressed as a function (e.g. $20n^2 + 10n + 5$) of the input size n.

Here the function $20n^2 + 10n + 5$ is an upper bound on the amount of time taken by the algorithm on any instance of input size n.

Running Time of an Algorithm

Our mathematical analysis based on the RAM model can only determine the running time of an algorithm on any real machine within a constant factor (accurate to a constant factor).

The exact amount of time depends on a particular real machine.

Thus there is no need to determine the exact amount of time taken by an algorithm on the RAM model.

Thus, we can say that the running time of the algorithm is proportional to n^2, instead of using $20n^2 + 10n + 5$. The difference between $20n^2 + 10n + 5$ and n^2 is a constant factor (≤ 45).

We use n^2 instead of $20n^2 + 10n + 5$ as n^2 is a simple function of n.

Big O Notation

This notation is used to give a simple upper bound on the running time of an algorithm.

Let the running time of an algorithm on any instance of size n be bounded from above by $T(n)$.

Let $f(n)$ be a simple function of n (e.g. n, n^2 (n times n), $n \log n$).

We say that $T(n)$ is $O(f(n))$ if there is an integer N and a constant c such that

$T(n) <= cf(n)$ for all $n \geq N$.

Note that we use mathematical induction to show that the above inequality holds for all $n \geq N$.

```
INSERTION-SORT(A)

    n = A.length
    for j = 2 to n  // j goes from 2, 3, ... up to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1..j-1].
      i = j - 1
      while  i > 0 and A[i] > key
         A[i+1] = A[i];
         i = i - 1
      A[i+1] = key;
```

Analysis of Insertion Sort

Let a constant d be an upper bound on the time (cost) of any line executed once.

The outer loop consists of three statements and a **while** loop with two statements in its body.

In one iteration of the outer loop, the **while** loop iterates at most $j-1$ times.

So the cost on the **while** loop is bounded by $3d \times (j-1) + d$.

Analysis of Insertion Sort

The worst-case ruuning time $T(n)$ of the algorithm is bounded by the following expression:

$T(n) \leq d + \sum_{j=2}^{n}[4d + (3d \times (j-1) + d)] + d$

$= 2d + \sum_{j=2}^{n}[5d] + 3d(\sum_{j=2}^{n}[j-1]$

$= 2d + 5d(n-1) + 3d[1 + 2 + ... + (n-1)]$

$= 2d + 5d(n-1) + 3d[\frac{(n-1)n}{2}]$

$\leq 2d + 5dn + 3dn^2 \leq 2dn^2 + 5dn^2 + 3dn^2 = 10dn^2$ for each $n \geq 1$.

So $T(n)$ is in $O(n^2)$.

The Divide-and-Conquer Algorithm Design Technique

In a recursive algorithm, we first check if an instance is small enough. If so, we obtain the solution to it in a straightforward manner. Otherwise, we solve it by following a divide-and-conquer strategy, which consists of three steps:

Divide: An instance of the problem is divided into a number of smaller instances of the same problem.

Conquer: Each smaller instance is solved recursively by calling the algorithm on it.

Combine: The solutions to the smaller instances are combined into the solution for the original instance.

Merge Sort

We use the divide-and-conquer technique to design an algorithm (called Merge Sort) for sorting a sequence of n items.

Divide: A sequence of n items is divived into two subsequenes of $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$ items, respectively.

Conquer: Each subsequene is sorted recursively.

Combine: The two sorted subsequenes are merged into one sorted sequence.

Note that $n = \lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor$. Here $\lceil x \rceil$ is the smallest integer $k \geq x$, and $\lfloor x \rfloor$ is the largest integer $k \leq x$.

So if n is even, then $\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor = \frac{n}{2} + \frac{n}{2} = n$.

If n is odd, i.e., $n = 2k + 1$ for some k, then $\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor = \lceil k + \frac{1}{2} \rceil + \lfloor k + \frac{1}{2} \rfloor = (k+1) + k = n$

MERGE-SORT

Let $A[1..n]$ be an array of n elements.

The array is sorted by calling MERGE-SORT(A, 1, n).

MERGE-SORT(A, p, r) sorts the elements in the subarray $A[p..r]$ by dividing $A[p..r]$ into $A[p..q]$ and $A[q+1..r]$, sorting each new subarray, and merging the sorted subarrays into one sorted subarray, where $q = \lfloor (p+r)/2 \rfloor$.

Let $m = r - p + 1$ be the size of $A[p..r]$. Then the size of $A[p..q]$ is $q - p + 1 = \lfloor (p+r)/2 \rfloor - p + 1 = \lfloor (p + r - 2p)/2 \rfloor + 1$

$= \lfloor (r-p)/2 \rfloor + 1 = \lfloor (r - p + 2)/2 \rfloor = \lfloor ((r - p + 1) + 1)/2 \rfloor$

$= \lfloor m + 1)/2 \rfloor = \lceil m/2 \rceil$

Let floor(x) denote $\lfloor x \rfloor$ in the following code.

```
// Sorts the elements in the subarray $A[p .. r]$
MERGE-SORT(A, p, r)
1  if p < r
2      q = floor( (p + r) / 2 )
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q+1, r)
5      MERGE(A, p, q, r)

// Takes the two sorted subarrays A[p .. q]
// and A[q+1 .. r], merges them into one sorted subarray,
// and saves it in the subarray A[p .. r].
MERGE(A, p, q, r)
1  n1 = q - p + 1  // the size of A[p .. q]
2  n2 = r - q      // the size of A[q+1 .. r]
3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
4  for i = 1 to n1    // copies A[p .. q] to L[1 .. n1]
5      L[i] = A[p + i - 1]
6  for j = 1 to n2    // copies A[q+1 .. r] to R[1 .. n2]
7      R[j] = A[q + j]
```

```
L[n1 + 1] = inf    // inf is greater than each element
R[n2 + 1] = inf
i = 1        // merges L[1 .. n1] and R[1 .. n2],
j = 1        // and saves the result in A[p .. r]
for k = p to r    // inf cannot be saved in A[p .. r]
    if L[i] <= R[j]
         A[k] = L[i]
         i = i + 1
    else A[k] = R[j]
         j = j + 1
```

```
A sequence of 6 elements to be sorted
Division
   9        2        5        8        2        3
p = 1, r = 6, q = floor( (1 + 6) / 2 ) = 3

 [ 9        2        5 ]    [ 8        2        3 ]
p = 1, r = 3, q = 2         p = 4, r = 6, q = 5

 [ 9        2 ]    [ 5 ]    [ 8        2 ]    [ 3 ]
```

```
p = 1              p = 3    p = 4             p = 6
r = 2              r = 3    r = 5             r = 6
q = 1                       q = 4

 [ 9 ]    [ 2 ]    [ 5 ]    [ 8 ]    [ 2 ]    [ 3 ]
 p = 1    p = 2             p = 4    p = 5
 r = 1    r = 2             r = 4    r = 5
```

Merge

```
 [ 9 ]    [ 2 ]    [ 5 ]    [ 8 ]    [ 2 ]    [ 3 ]
 [ 2        9 ]    [ 5 ]    [ 2        8 ]    [ 3 ]
 [ 2        5        9 ]    [ 2        3        8 ]
 [ 2        2        3        5        8        9 ]
```

Analysis of MERGE-SORT

We can show that the algorithm is correct by formulating and proving loop invariants. The proof is omitted.

Let the running time of MERGE-SORT on a sequence of n elements be bounded from above by $T(n)$.

Based on the above analysis, we obtain the following recurrence for $T(n)$ for some constant $d > 0$.

$T(1) = d$,
$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + dn$ if $n > 1$.

Analysis of MERGE-SORT

Assume that n is a perfect power of 2, i.e., $n = 2^k$ for some $k \geq 0$.

Then we have
$T(n) = 2T(n/2) + dn = 2[(2T(n/2^2)) + d(n/2)] + dn$
$= [2^2 T(n/2^2) + dn] + dn = 2^2 T(n/2^2) + 2dn$
$= 2^h T(n/2^h) + hdn$ for $1 \leq h \leq k$
$= 2^k T(n/2^k) + kdn = 2^k T(1) + kdn = nd + kdn = dn(k+1)$
$= dn(1 + \log n) \leq dn(\log n + \log n) = 2dn \log n = cn \log n$, if $n > 2$
with $c = 2d$.

Analysis of MERGE-SORT

If n is not a perfect power of 2, then we have $2^k < n < 2^{k+1}$ for some k.

Then we have
$T(n) \leq T(2^{k+1}) \leq c2^{k+1}(k+1) = 2c2^k(1+k)$
$\leq 2cn(1 + \log n) \leq 2cn(\log n + \log n) = 4cn \log n$, if $n > 2$.

Thus, $T(n)$ is in $O(n \log n)$.