Worst Case & Best Case Scenarios

• Sometimes it is informative to consider what is the worst (or best) case scenario for your algorithm.

• It could be the case that your algorithm performs well on most instances of the input but has a few pathological inputs on which it is very slow. However, in general, this will not be the case.

• For example consider a scalar array of length N which we want to search to see if any element is equal to a given value, say 17. If the first element in the array happens to be 17, then the algorithm is complete in one step (best case scenario) but if the last element, or no element, is 17 then we have to check all N elements so we will perform N comparisons. We say that this "exhaustive search" is linear in N even though there may be some instances of input where it performs faster.

Examples of calculating a formula for the growth rate.

Given two n-vectors $\vec{u}$ and $\vec{v}$, the scalar dot product is denoted by

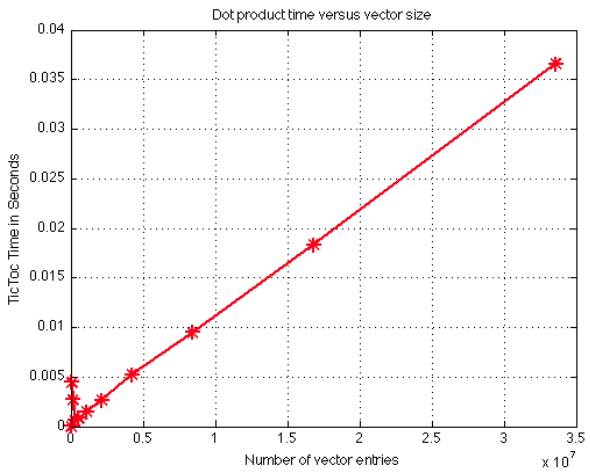
$$\vec{u}^T\vec{v} = \vec{u}\cdot\vec{v} = \sum_{i=1}^{n} u_i v_i$$

where u_i denotes the ith entry of the vector $\vec{u}$.

This can be computed in approximately n operations:

- 1 initialization and $2n$ "fetches" from memory
- n multiplies
- $n-1$ adds
- 1 write to memory

If we count only the n+(n-1) computational operations, we have $2n-1$ operations or a linear algorithm, i.e., $\mathcal{O}(n)$ algorithm.



Plot of the time to compute a dot product versus the size of the vector. Clearly the growth is linear in the size of the vector because as the size is doubled, the work is also doubled.

Shortest path

Suppose we have N cities, and we are interested in determining the shortest driving time st**(**i, j**)** to drive from each city i to each city j.

• We assume that we start with a table that gives the driving time dt**(**i, j**)** for a direct trip from city i to each city j.

• If there is a direct route from city i to city j then it is easy. However many cities may not have a direct link. Usually there are many routes from one city to another and we want to find the shortest of all possible routes.

• Between city i and city j there are N **−2** other cities, so theoretically there are **(**N **− 2)!** routes to check for each city combination. This seems like an O(N!) problem, also known as "impossible"!

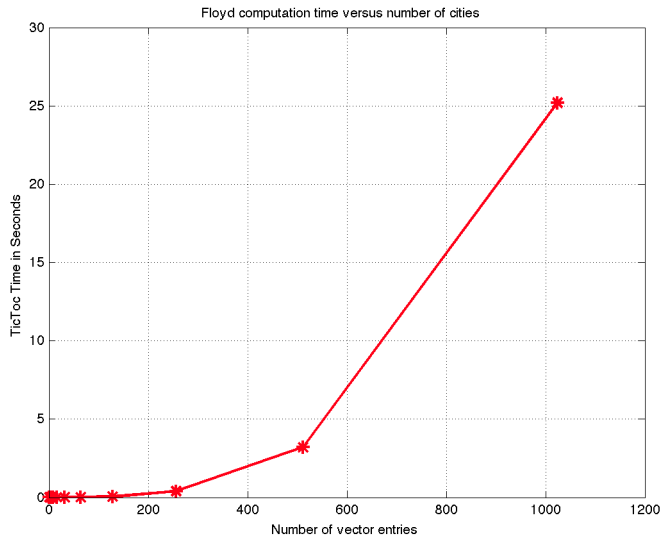
Floyd's algorithm for shortest path problem

Instead of being impossible, Floyd's algorithm shows a simple way to compute

the entire table of possible distances in just a few lines of code:

```
set st = dt
for k = 1 : n
    for j = 1 : n
        for i = 1 : n
            st(i,j) = min ( st(i,j), st(i,k) + st(k,j) )
        end
    end
End
```

Don't worry about why this algorithm works right now but simply calculate the work required. What is the growth as a factor of **n**?



Plot of time versus number of cities for Floyd's algorithm.

Brute Force Algorithms

• These are algorithms which take a straightforward and often the most obvious approach to solving a problem.

• The basic idea is often *to try all possibilities and see if any of them works.*

• These algorithms are rarely called clever or efficient but should not be overlooked as an important design strategy.

• This approach is applicable to a very wide range of problems.

• Sometimes we only need to solve a small problem for an educational purpose or to verify some theoretical result and in this case a brute force approach may be the quickest to implement.

Example

Determine the greatest common divisor (gcd) of two integers, **m**, **n.** for example, determine gcd(54, 99)

• A brute force approach to determining this would be to check consecutive integers; e.g., check 54, then 53, then 52, etc. until we find the largest that divides both numbers.

• **How would we implement such a method?**

We could start with 2 and increase our test divisor by one until we reach either **m** and **n** (the smallest one) but it would probably be better to start with the largest possible divisor and decrease.

– We know that the gcd has to be <= min{**m, n**}.

– So we set our guess for the gcd to be **t** = min{**m, n**}.

– If **t** divides both **m** and **n** (i.e., the remainder is zero) we are done;

– If the remainder is not zero (for either **m** or **n**) then we reduce **t** by one and continue

Consecutive integer checking algorithm:

Input: two integers, m and n

Output: integer t which is gcd(m, n)

Step 1. Set t = min**{**m, n**}**

Step 2. Divide m by t; if the remainder is 0, go to Step 3; otherwise go to Step 4.

Step 3. Divide n by t; if the remainder is 0, return the value of t as the gcd; otherwise go to Step 4.

Step 4. t = t − 1; go to Step 2

This is a description of the code but it is not really written in pseudocode format. However, it is a format that is often used in books and papers.

Example gcd(16,36)

$t = \min\{16, 36\} = 16$

$t = 16$ 16/16 has remainder 0 , 36/16 does not have remainder 0

$t = 15$ 16/15 does not have remainder 0

$t = 14$ 16/14 does not have remainder 0

$\vdots$

$t = 8$ 16/8 has remainder 0, 36/8 does not have remainder 0

$t = 7$ 16/7 does not have remainder 0

$\vdots$

$t = 5$ 16/5 does not have remainder 0

$t = 4$ 16/4 has remainder 0 36/4 has remainder 0; return gcd=4

Of course this is definitely not the most efficient approach to finding the greatest common divisor. The worst case scenario would be when we have to check all numbers from $\min\{m, n\}$ to 2. At each step we have to do one or two divisions so the work for the worst case scenario is $< 2\min\{m, n\}$ so it is linear.

Sorting a List

Suppose we have a list of n orderable items (names, numbers, etc.) and we want to sort these based upon some criteria. Dozens of algorithms have been developed to perform such a task. Clearly it is a task that is prevalent today; e.g., sorting a list of students by GPA, sorting a list of employees by years of service, ordering a list of items such as TVs that you want to purchase by price, etc.

You may already know some methods to do this, but for now, pretend you don't and let's look at a couple of brute force approaches. We want a straightforward approach but remember what one person may view as straightforward, another may not so we consider two candidates here.

For simplicity of exposition, we will assume that we are sorting a list of n numbers in ascending order.

Selection Sort Algorithm

This algorithm works by putting the smallest entry in the first position of the array, then putting the second smallest in the second position, etc.

- Scan list to find smallest entry and exchange first entry of list with this smallest entry.
- Scan second through n entries in list to find smallest entry and exchange this with the second entry.
- Scan third through n entries in list to find smallest entry and exchange this with the third entry.
- Continue until you are scanning entries $n-1$ through n to find the smallest entry and exchange it with $(n-1)$st entry
- The result is the sorted list.
- An equivalent algorithm would be to start with scanning the array to find the largest entry and putting it in the nth position, then the second largest in the $(n-1)$st entry, etc.

Selection sort for real array

Input: array a(1:n) of numbers and its length n
Output: the array a(1:n) sorted in ascending order

```
for i=1, n-1
    min loc = i
    for j=i+1, n
        if ( a(j) < a(min loc) ) min loc = j
    end for loop over j
    swap a(i) and a(min loc)
end for loop over i
```


Example

Apply the Selection Sort algorithm to the array of numbers
(49, 61, 19, 12)

For the first sweep we locate the smallest entry in the entire array (the fourth entry) and exchange it with the first entry to get
(12, 61, 19, 49.)

For the second sweep we locate the smallest entry in positions 2 through 4 (the third entry) and exchange it with the second entry to get (12, 19, 61, 49).

For the third and final sweep we find the smallest entry in positions three and four (the fourth entry) and exchange to get (12, 19, 49, 61).
The algorithm is complete.

How much work does this algorithm take?

- Clearly the amount of work depends upon the length of the array n. We want to determine precisely how it depends upon n.
- For determining formulas for the work the following results from calculus are useful.

$$\sum_{i=1}^{m} i = \frac{m(m-1)}{2} \qquad \sum_{i=1}^{m} i^2 = \frac{m(m+1)(2m+1)}{6}$$

- The key work that has to be done is the comparison of two elements of the array. Looking at our algorithm description we see that the outer loop is from 1 to $n-1$ and the inner loop is from $i+1$ to n and we have to do one comparison in the inner loop. Consequently we have

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1}\Big[n-(i+1)+1\Big] = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$$

$$= n\sum_{i=1}^{n-1} 1 - \sum_{i=1}^{n-1} i = n(n-1) - \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

- So we say the algorithm is quadratic in n and is $O(n^2)$.
- Of course we have to swap elements but this is only done n – 1 times.
- Recall that an algorithm which has quadratic growth increases the work by a factor of four when n is doubled.

Bubble sort

- Movie
- http://www.youtube.com/watch?v=lyZQPjUT5B4

A second brute force approach to sorting is the **Bubble Sort** which gets its name from the fact that the largest entry "bubbles up" to the top. Recall that Selection sort started by finding the smallest entry. In the first sweep of Bubble sort the largest entry is moved until it reaches the last position in the array. In the next sweep the second largest entry makes its way to the n − 1 position, etc.

• In the first sweep getting the largest entry to the last position is accomplished by first checking the first and second entries; if the first is larger than the second then they are interchanged.

• Next, the second and third entries are checked and if the second is larger than the third then they are interchanged; if not, then nothing is done.

• This continues until the (n − 1)st and nth entries are compared and interchanged if the (n − 1)st is larger than the nth entry; the first sweep is completed.

• Then one starts over but we only have to compare entries in the first through (n − 1)st components because we have already moved the largest component to the last entry. This procedure is continued until the entire array is sorted.

Bubble Sort for real array

Input: array a(1:n) of numbers and its length n
Output: the array a(1:n) sorted in ascending order

```
for i=1, n-1
    for j=1,n-i
        if ( a(j+1) > a(j ) ) swap a(j) and a(j+1)
    end for loop over j
end for loop over i
```

Example

Apply the Bubble Sort algorithm to the array of numbers (49, 61, 19, 12)
For the first sweep we have the following steps
49 < 61 so do nothing (49, 61, 19, 12)
61 > 19 so interchange to get (49, 19, 61, 12)
61 > 12 so interchange to get (49, 19, 12, 61)

For the second sweep
49 > 19 so interchange to get (19, 49, 12, 61)
49 > 12 so interchange to get (19, 12, 49, 61)

Note that we do not have to compare the third and fourth entries because in the first sweep we have moved the largest entry to the fourth position.

For the third sweep
19 < 12 so interchange to get (12, 19, 49, 61)

Note that we do not have to compare the second and third or third and fourth entries because in the first sweep we have moved the largest entry to the fourth position and in the second sweep we have moved the second largest to the third position.
Algorithm is complete.

How much work does this algorithm take?

Remember that the Selection Sort Algorithm took $\mathcal{O}(n^2)$ operations. It turns out that the Bubble Sort Algorithm takes the same amount of work. We have

$$\sum_{i=1}^{n-1}\sum_{j=1}^{n-i} 1 = \sum_{i=1}^{n-1}(n-i) = n\sum_{i=1}^{n-1} - \sum_{i=1}^{n-1} i$$

$$= n(n-1) - \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

and thus the algorithm is $\mathcal{O}(n^2)$.

Sequential Search

• Suppose that we want to search elements in a list or array with a given value called a **search key**. For example, we might want to find the element in an array that equals 17 or 'Tallahassee'.

• The brute force approach is to be given a list say a and a search key say K.

– Check if a(1) = K; if so terminate, otherwise continue.
– Check if a(2) = K; if so terminate, otherwise continue.
– Continue until one finds i such that a(i) = K or the list is exhausted

Sequential Search Algorithm

Input: an array **a**(1 : **n**) and a search key K

Output: the index of the first element of **a** that matches K or 0 if no match

i=0
while i < n and a(i) ≠ K do
    i ← i+1
if i < n return i
else return 0

As we discussed last time, the worst case scenario is that we have to check all **n** elements in the array so we have linear growth O(n) , whereas the best case scenario is O(1) when the first entry of the array equals the key.

Exhaustive Searches

This brute force approach determines all possible combinations of every feasible solution and picks the one which satisfies the given criteria. This approach is impractical for all but the smallest problems because the work is **n!**.

Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is to find the shortest tour through **n** cities with known distances between them. It was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The brute force approach/exhaustive search would be to find **all possible routes and then pick the shortest**.

It has applications in planning, logistics, microchip design and even DNA sequencing.

Example Consider 4 cities A, B, C, and D and suppose we are given the following direct distances between cities which we denote, e.g., **d(A,B)**. Use the

brute force approach to find the minimum distance to travel to all cities if we have the constraint that we want to start and end at city A.

$$d(A,B) = 10 \quad d(A,C) = 70 \quad d(A,D) = 110$$
$$d(B,C) = 40 \quad d(B,D) = 60 \quad d(C,D) = 30$$

We determine the distances for all possible routes and take the smallest

$A \to B \to C \to D \to A$ =10+40+30 +110=190

$A \to B \to D \to C \to A$=10+ 60+30+70=160

$A \to C \to D \to B \to A$=70+30+60+10=170

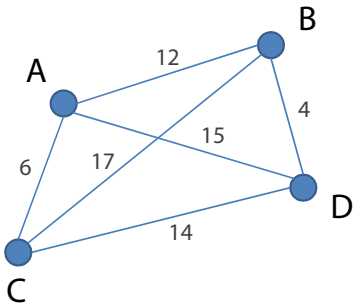
$A \to C \to B \to D \to A$=70+40+60+110=280

$A \to D \to B \to C \to A$=110+60+40+70=280

$A \to D \to C \to B \to A$=110+30+40+10=190

So the shortest path is $A \to B \to D \to C \to A$.

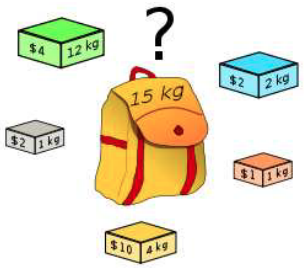
Traveling salesman problem



A, B, C ,D, A = 58
A, B, D, C, A = 36
A, C, B, D, A = 42
A, C, D, B, A = 36
A, D, B, C, A = 42
A, D, C, B, A = 58

Knapsack Problem

In this problem, we are given a set of items, each with a weight and a value and we want to determine the number of each item to include in a collection so that the total **weight is less** than or equal to a given limit and the total value is **as large as possible.**



In the example illustrated we are trying to keep the total weight under 15 kg while maximizing the dollar amount.

As with the traveling salesman problem, the brute force approach/ exhaustive search is to find all possible combinations which are feasible ones (within the restriction given on total weight) and choose the one which has the largest value.

For example, for the case illustrated we look at all possible combinations such as

1 green = 12 kg and $4 value

1 green + 1 blue = 14 kg and $6 value

1 green + 1 blue + 1 red = 15 kg and $7 value

1 green + 1 blue + 1 grey = 15 kg and $8 value

1 green + 1 red = 13 kg and $5 value

etc.

Knapsack problem

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

Capacity(K) = 5

subset	weight	value
{1}	2	$12
{2}	1	$10
{3}	3	$20
{4}	2	$15
{1,2}	3	$22
{1,3}	5	$32
{1,4}	4	$27
{2,3}	4	$30
{2,4}	3	$25
{3,4}	5	$35
{1,2,3}	6	not feasible
{1,2,4}	**5**	**$37**
{1,3,4}	7	not feasible
{2,3,4}	6	not feasible
{1,2,3,4}	8	not feasible

Example

Suppose our limit to the weight of the knapsack is 10 kg. We have four items

Item #1 weighs 7 kg and has a value of $42

Item #2 weighs 3 kg and has a value of $12

Item #3 weighs 4 kg and has a value of $40

Item #4 weighs 5 kg and has a value of $25

Make a table of all possible combinations, their weight and total value; then determine the solution.

7 + 3 = 10

3 + 4 = 7

3 + 5 = 8

4 + 5 = 9

??

42$+12$=54$

12$ + 40$ = 52$

12$ + 25$ = 37$

40$ + 25$=65$

If a combination weighs more than 10 kg indicate that it is not feasible.

Divide and Conquer Algorithms

A popular approach to algorithm design is divide and conquer. The basic idea is to

- divide the problem into several smaller problems of the same type where ideally the smaller problems are of the same size;
- solve each smaller problem;
- combine solutions of smaller problems to form desired solution.

Divide and conquer algorithms are ideally suited for parallel computations.

As an example, consider the problem of summing 100 numbers $\mathbf{a_1}$, $\mathbf{a_2}$, . . . , $\mathbf{a_{100}}$.
The brute force approach is, of course, to add $\mathbf{a_1}$ and $\mathbf{a_2}$ then add the result to $\mathbf{a_3}$, etc. A divide and conquer approach might be to sum the first fifty numbers, $\mathbf{a_1}$, . . . , $\mathbf{a_{50}}$ and then sum the last fifty numbers $\mathbf{a_{51}}$, . . . , $\mathbf{a_{100}}$ and then add the
result of summing the first fifty numbers and the last fifty numbers.

$$\alpha = a_1 + a_2 + \cdots + a_{50} \qquad \beta = a_{51} + a_{52} + \cdots + a_{100}$$

$$\text{answer} = \quad \alpha + \beta$$

However, there doesn't appear to be any advantage for this approach compared to the brute force approach (on a serial machine). So not every divide and conquer algorithm is more efficient than a brute force approach.

However, there are divide and conquer algorithms which are more efficient than brute force approaches.

Sorting Algorithms using Divide and Conquer

We saw two brute force approaches to sorting an array – Selection Sort and Bubble Sort. Both algorithms were $\mathcal{O}(n^2)$. We now want to look at two important sorting routines which take the divide and conquer approach and are $\mathcal{O}(n \log n)$.

MergeSort

The basic idea is simple.

- We divide the array $a(1:n)$ into two smaller arrays $a(1:n/2)$, $a(n/2{+}1, n)$.
- Each of the two smaller arrays is divided again; continue this procedure until you have arrays of length one.
- Merge smaller arrays into a sorted array of length n.

- http://www.youtube.com/watch?v=XaqR3G_NVoo

88

As an example consider the array

$$(45, 12, 61, 19, 71, 22, 4, 33)$$

We divide it into two arrays

$$(45, 12, 61, 19) \qquad (71, 22, 4, 33)$$

Now each of these arrays of length 4 is divided into two arrays of length two

$$(45, 12) \quad (61, 19) \qquad (71, 22) \quad (4, 33)$$

and finally we have

$$(45) \quad (12) \qquad (61) \quad (19) \qquad (71) \quad (22) \qquad (4) \quad (33)$$

We merge each array to form sorted arrays of length two

$$(12, 45) \quad (19, 61) \qquad (22, 71) \quad (4, 33)$$

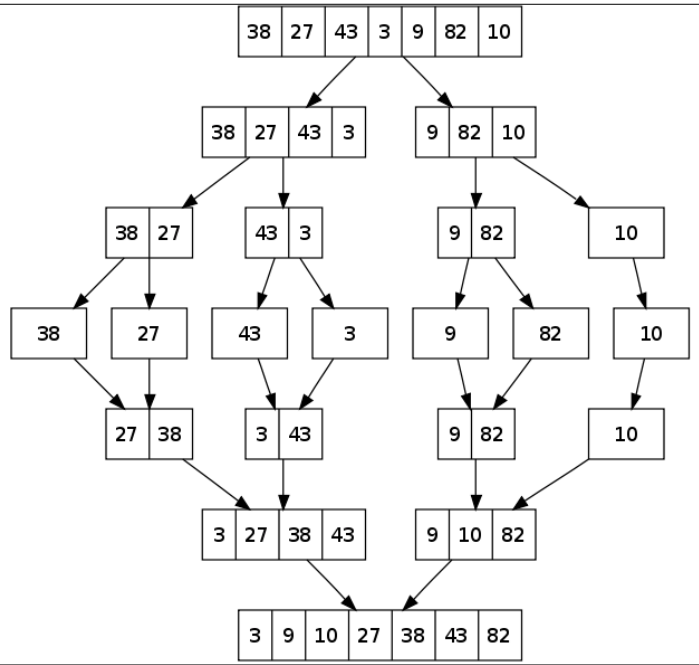
Now we continue to reassemble the array by merging to form two sorted arrays of length 4

$$(12, 19, 45, 61) \qquad (4, 22, 33, 71)$$

and finally merge these two sorted arrays to form the final sorted array

$$(4, 12, 19, 33, 45, 61, 71)$$

89



90

Lets consider the last step where we want to merge the two sorted arrays of length 4 above. Let u = (3,10,23,54), v = (1,5,25,75). We set up an array of length 8, call it w, for the merged array.

• Lets use the pointer i to indicate the next u value to select; j to indicate the next v value to select and k to indicate the next w value to fill. Initially

i = j = k = 1.

• At each step we check:

– if u(i) <= v(j) then we set w(k) = u(i) and increment i, k;

– otherwise w(k) = v(j) and increment j, k;

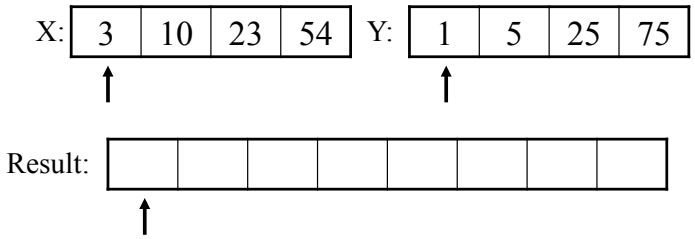
91

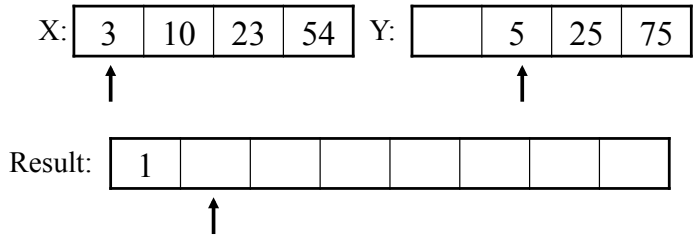
Merge sort

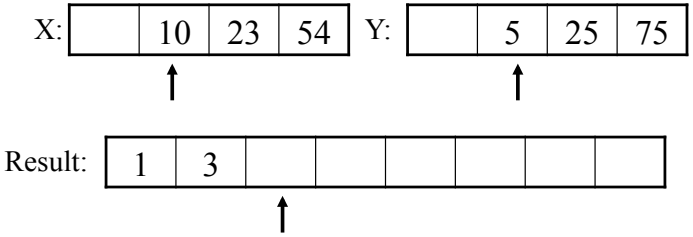
Input: u,v, two sorted arrays of length n

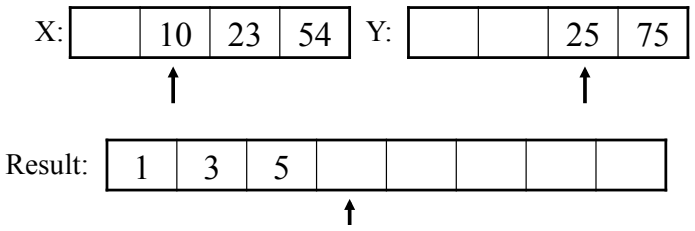
Output: w, an array of length 2n which is the sorted array formed by merging u and v

```
Set i=j=k=1
while i< n and j<n
    if u(i) < v(j)
        w(k) ← u(i); k ← k+1; i ← i+1
    else
        w(k) ← v(j); k ← k+1; j ← j+1
end while
if i=n
    copy v(j:n) into w(k:2n)
else
    copy u(i:n) into w(k:2n)
```

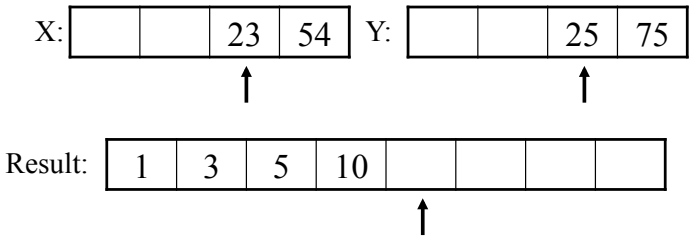





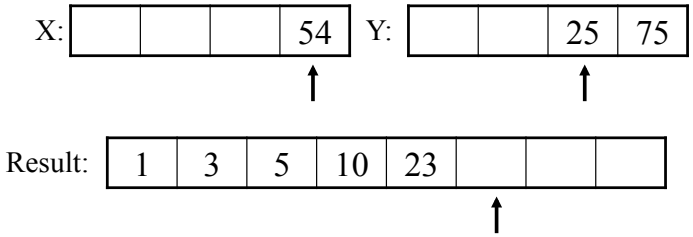



X:
10
23
54
Y:
25
75
Result:
1
3
5

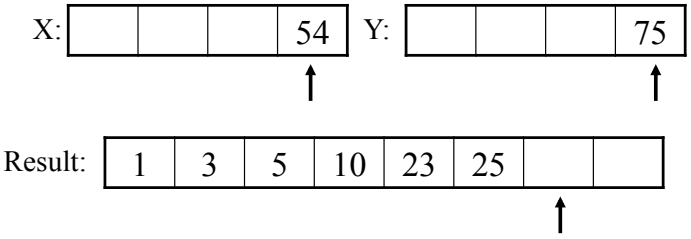
96


X:
23
54
Y:
25
75
Result:
1
3
5
10

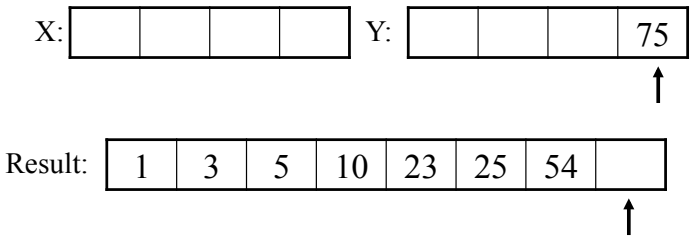
97


X:
54
Y:
25
75
Result:
1
3
5
10
23

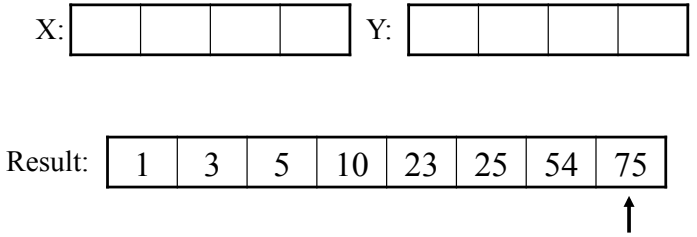
98


X:
54
Y:
75
Result:
1
3
5
10
23
25

99



100



101

We can summarize the steps before the merge in the table below.

Level	Problem Size	# Problems
0	8	1
1	4	2
2	2	4
3	1	8

Before we present the algorithm and argue that it is indeed more efficient than Selection Sort or Bubble Sort we need to clarify how to perform the merge.

The shortcoming of this approach is that the merge requires an extra array of length **n**.

102

Note that we could easily modify this routine so that the input arrays had different lengths.

Why do we think that Merge sort is more efficient that Selection or Bubble sort which are $\mathbf{O(n^2)}$. Recall that if an algorithm if $\mathbf{O(n^2)}$ then when **n** is doubled, then the work is increased by a factor of 4. Is that the case for Merge sort?

Consider the example we had an array of length 8; we divided it into arrays of length 4 then of length 2 and finally of length 1 and then merged the arrays of length 2 and finally the arrays of length 4. What if our original array was of length 16? Then basically we have to first divide into two arrays of length 8 and then proceed as before except we have one additional merge – the two arrays of length 8. So when **n** is doubled we do not increase the work by a factor of 4 but rather we simply added one more level of work. This is indicative of logarithmic growth.

103

T(n) = 2 T(n/2) + n
= 2 (2T(n/4) +n/2) + n
=4T(n/4)+n+n
=4T(n/4)+2n
=4 (2T(n/8) +n/4) + 2n
=8T(n/8)+n+2n
=8T(n/8)+3n
$=2^k T(n/2^k) + kn$ $n=2^k \rightarrow \log(n)=k$
= nT(1) +n logn
= n+ n log n
O(n log (n))

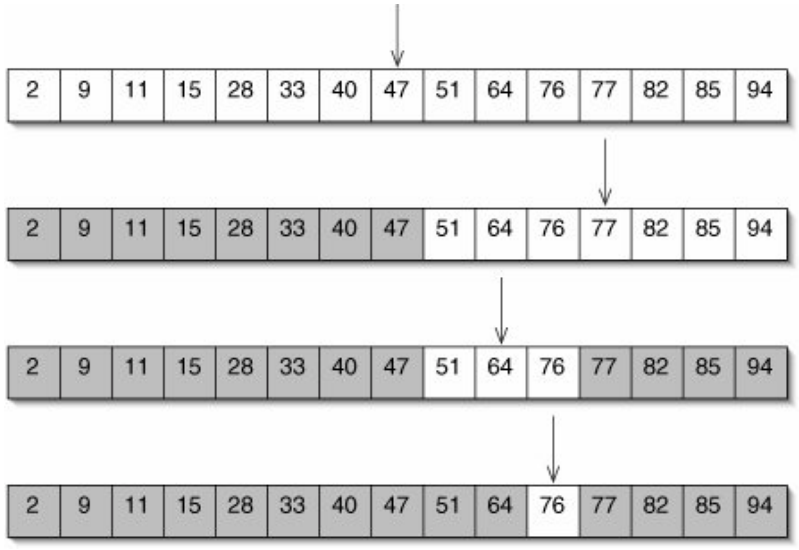
Binary Search

Suppose you have an array and you want to search with a key $\mathcal{K}$. The brute force or sequential approach is to check the first entry, then the second, then the third, etc. until you have found the desired entry.

However, if the list is sorted then we can use this fact to create a more efficient sort routine. If you had an unsorted array which you need to search many times (such as a phone book) it is advantageous to first sort the array and then use a more efficient search algorithm than sequential search.

Binary Search has some similarity to the Bisection Method which you studied for finding the roots of a function $f(x)$ in $[a, b]$ where $f(a)f(b) < 0$.

Suppose we are given an array $a(1 : n)$ already sorted in ascending order to search using the key $\mathcal{K}$.



- Check if $\mathcal{K} > a(n)$ or $\mathcal{K} < a(1)$ then not in array.
- Set $i_L = 1$, $i_R = n$.
- First check the middle value of the list, say $m = n/2 = (i_L + i_R)/2$. If $a(m) = \mathcal{K}$ then we are done; if $a(m) < \mathcal{K}$ then $\mathcal{K}$ must be in the smaller list $a(m : n)$ so set $i_L = m$; otherwise it is in $a(1 : m)$ so set $i_R = m$. We now know that $\mathcal{K} \in a(i_L, i_R)$.
- Set $m = (i_L + i_R)/2$. (Recall that in Matlab we have to make sure this is an integer; the correct Matlab command is `m=floor( iL+iR )/2`.) If $a(m) = \mathcal{K}$ then we are done; if $a(m) < \mathcal{K}$ then $\mathcal{K}$ must be in the smaller list $a(m : n)$ so set $i_L = m$; otherwise it is in $a(1 : m)$ so set $i_R = m$. We now know that $\mathcal{K} \in a(i_L, i_R)$.
- Continue in this manner until $\mathcal{K}$ is found.

Binary Search Input: sorted array a of length n, search key $\mathcal{K}$

Output: index of the array element $= \mathcal{K}$ or 0 if not in array

```
if K < a(1) or K > a(n) return 0
left=1; right =n
while left ≤ right do
    m=(left+right)/2
    if a(m)=K return m
    if a(m) > K
      set right=m
    else
      set left = m
end while
```


Example Use Binary Search to search the array

$$a = \{5, 9, 12, 17, 21, 45, 81, 109, 122\}$$

for the element 17.

- set $i_L = 1$, $i_R = 9$ and $m = 5$
- 17< $a(5) = 21$ so set $i_R = 5$; key is in $a(1:5)$
- $m = (i_L + i_R)/2 = 3$
- $17 > a(3) = 12$ so set $i_L = 3$; key is in $a(3, 5)$
- $m = (i_L + i_R)/2 = (3 + 5)/2 = 4$
- $17 = a(4)$ so we are done; return 4

What about the efficiency of Binary Search? Is it **O(n)**? Recall that if it is **O(n)** then when we double **n** the work should be increased by two. However, in this case if we double the length of the array we only increase the work by one

level which is indicative of logarithmic growth. One can show that the method is **O(log n)**.

$n=2^k$ → $\log(n)=k$

$$T(n) = \begin{cases} T(\frac{n-1}{2}) + 1 & \text{if } n \geq 2; \\ 1 & \text{if } n = 1. \end{cases}$$

Multiplying Large Integers

- Some applications, such as modern cryptology require multiplying integers which are over 100 digits long. These integers are too long to fit into a single word of a computer so they require special treatment.
- What is the brute force approach (the usual method we were taught in elementary school) to multiplying two integers A and B of length n? We simply take the first digit of A and multiply it by all n digits of B (n multiplications). Then we take the second digit of A and multiply it by all n digits of B. Continuing in this manner we see that we have n^2 multiplications followed by fewer **(**n – **1)** additions so the method is O**(**n^2**)**.
- Can we design an algorithm which has fewer operations than O**(**n^2**)**? The answer is yes, using the Divide and Conquer strategy.
- The easiest way to see how to do this is to look at an example.

Example Multiply 29 by 13 (=377) using a Divide and Conquer approach.

We first note that

$$29 = 2 * 10^1 + 9 * 10^0 \qquad 13 = 1 * 10^1 + 3 * 10^0$$

so that

$$29 * 13 = \left(2 * 10^1 + 9 * 10^0\right) * \left(1 * 10^1 + 3 * 10^0\right)$$

$$= (2*1) * 10^2 + (9*3) * 10^0 + (9*1 + 2*3) * 10^1 = 200 + 27 + 150 = 377$$

But if we multiplied the two numbers by the usual approach we would have 4 multiplications and that's exactly what we have here!

The idea is to compute the coefficient (9*1+2*3) of 10^1 by taking advantage of the two multiplications we have already done which are 2*1 and 9*3; if we can do the computation 9*1+2*3 in one multiplication then we have improved upon the brute force approach. We note that this can be done if we write the

$$(9 * 1 + 2 * 3) = (9 + 2) * (1 + 3) - (2 * 1) - (9 * 3)$$

Now because we have already computed 2*1 and 9*3 we are only performing one multiplication but of course we have added some additions.

In general, if we have two two-digits numbers $a = a_1a_0$, $b = b_1b_0$ then

$$c = a * b = c_2 * 10^2 + c_1 * 10^1 + c_0 * 10^0$$

where $c_2 = a_1 * b_1$ (the product of the tens digits), $c_0 = a_0 * b_0$ (the product of the ones digits) and $c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_0 + c_2)$, the product of the sum of the digits minus c_0 and c_2 which were previously computed.

Where is the divide and conquer strategy in this algorithm?

Well, it's not there yet! We want to use this idea of multiplying two two-digit integers to integers with more digits.

Suppose we want to multiply two 6-digits integers,

$$a = a_5a_4a_3a_2a_1a_0 \qquad b = b_5b_4b_3b_2b_1b_0$$

We now divide each in half (here's the divide part)

$$\alpha_1 = a_5a_4a_3 \quad \alpha_0 = a_2a_1a_0 \qquad \beta_1 = b_5b_4b_3 \quad \beta_2 = b_2b_1b_0$$

The resulting product $a * b$ can be formed using the ideas above

$$c = a * b = (\alpha_1 * 10^3 + \alpha_0) * (\beta_1 * 10^3 + \beta_0)$$

$$= (\alpha_1 * \beta_1) * 10^6 + (\alpha_1 * \beta_0 + \alpha_0 * \beta_1) * 10^3 + (\alpha_0 * \beta_0)$$

$$= c_2 * 10^6 + c_1 * 10^3 + c_0$$

where c_2 is the product of their first halves; c_0 is the product of their second halves and $c_1 = (\alpha_1 + \alpha_0) * (\beta_1 + \beta_0) - (c_2 + c_0)$ as before.

If $n/2$ is even (not in this case) we can apply the algorithm recursively until the integers are deemed small enough to multiply in the usual way.

Example Use the Divide and Conquer approach to multiply

$$4127 * 3456 = 14,262,912.$$

$4127 = 41 * 10^2 + 27, 3456 = 34 * 10^2 + 56$

$4127 * 3456 = \left(41 * 10^2 + 27\right) * \left(34 * 10^2 + 56\right) = (41 * 34) * 10^4 + (27 * 34 + 41 * 56) * 10^2 + \left(27 * 56\right)$

The cross term is computed as $(27 * 34 + 41 * 56) = (41 + 27) * (34 + 56) - 41 * 34 - 27 * 56$

We apply the algorithm recursively to compute the products 41*34, 27*56 and 68*90 and then substitute into the formula

(41 * 34) * 10^4 + (68 * 90–41 * 34 – 27 * 56) * 10^2 + (27 * 56)

To form 41*34 we write $41 * 34 = (4 * 10^1 + 1) * (3 * 10^1 + 4) = 4 * 3 * 10^2 + 4 * 1 + (1 * 3 + 4 * 4) * 10^1 = 1200 + 4 + (1 * 3 + 4 * 4) * 10^1$. Again the cross term is written as $(1 * 3 + 4 * 4) = (1 + 4) * (3 + 4) - 4 - 12 = 5 * 7 - 16 = 19$. Thus $41 * 34 = 1204 + 19 * 10^1 = 1204 + 190 = 1394$.

Similarly $27 * 56 = 1512$ and 68*90=6120.

We now return to our formula and substitute these values in $(41*34)*10^4+(68*90-41*34-27*56)*10^2+\left(27*56\right) = 1394*10^4+(6120-1394-1512)*10^2+1512 = 13,940,000+3214(100)+1512 = 13,941,512+321,400 = 14,262,912$

Matrix Multiplication

Suppose we want to multiply two $n \times n$ matrices A and B. The standard way we have learned to do this, is to dot each row of A into each column of B. For each dot product of a row and column we perform n multiplications and $(n-1)$ additions. So when we dot the first row of A into all n columns of B we have n^2 multiplications and $n(n-1)$ additions. Now there are n rows of A to use so we have $n(n^2)$ multiplications and $n(n(n-1))$ additions. Consequently the method grows with n like n^3.

Is it possible to obtain an algorithm that does it in less than $\mathcal{O}(n^3)$? The answer is actually yes; the approach parallels that of the integer multiplication. The first algorithm to be developed was the Strassen Matrix Multiplication algorithm (1969) which is approximately $\mathcal{O}(n^{2.8})$; there are modifications to it that have a growth rate of $\mathcal{O}(n^{2.376})$. The algorithms are not widely used because there is some instability for some matrices.

Strassen's algorithm is an application of Divide and Conquer strategy. We will just look at the result (similar to the one of multiplying two 2-digits integers) which allows us to perform less multiplications. The algorithm will be applied recursively as we did with integer multiplication.

The algorithm is based upon the following observation about multiplying two 2×2 matrices, A, B with entries a_{ij}.

$$C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

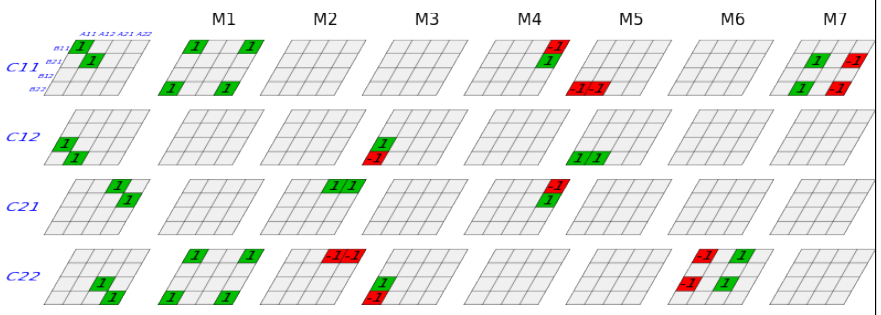
$$= \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

where

$$m_1 = (a_{11} + a_{22}) * (b_{11} + b_{22})$$

$$m_2 = (a_{21} + a_{22}) * b_{11}, \qquad m_3 = (b_{12} - b_{22}) * a_{11} \qquad m_4 = (b_{21} - b_{11}) * a_{22}$$

$$m_5 = (a_{11}+a_{12})*b_{11}, \qquad m_6 = (b_{11}+b_{12})*(a_{21}-a_{11}) \qquad m_7 = (b_{21}+b_{22})*(a_{12}-a_{22})$$



Thus there are 7 multiplications required instead of the usual 8. Not much of a savings but we wouldn't use the algorithm to multiply 2×2 matrices. As n goes to infinity it is asymptotically faster than the straightforward approach.

If we have two 4×4 matrices to multiply then we divide them into 2×2 blocks and use the approach above. If the matrices are of an odd dimension then we can pad with a row of zeros.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

where each A_{ij}, B_{ij}, C_{ij} is a 2×2 block.

$$O([7 + o(1)]^n) = O(N^{\log_2 7 + o(1)}) \approx O(N^{2.8074})$$

Decrease and Conquer Algorithms

The next design strategy we encounter is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.

For example, consider again the problem of computing a^n for a given scalar a and integer n. This is the given instance of the problem with n specified. We now reduce it to a smaller instance of the same problem. One obvious way is to write

$$a^n = \left[a^{n/2}\right]^2$$

Of course this only works if n is even. If n is odd, then $(n-1)$ is even so we write a^n as

$$a^n = aa^{n-1} = a\left[a^{(n-1)/2}\right]^2$$

So to summarize, we apply the strategy recursively and use the formula

$$a^n = \begin{cases} \left[a^{n/2}\right]^2 & \text{if } n \text{ is even} \\ \left[a^{(n-1)/2}\right]^2 & \text{if } n \text{ is odd and } > 1 \\ a & \text{if } n = 1 \end{cases}$$

In this case we have decreased the problem by a constant each time with constant $1/2$ when n is even.

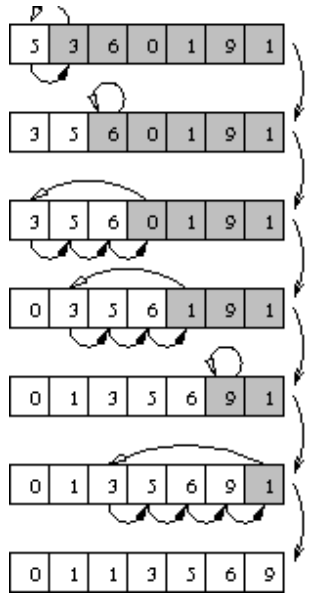
Example:

$\pi^8 = \pi^4\ \pi^4$
$\pi^4 = \pi^2\ \pi^2$
$\pi^2 = \pi\ \pi$

Insertion Sort

- This sorting routine is an example of the paradigm to decrease the size by a constant (one in this case) whereas in the previous example we reduced the problem by a factor (1/2 in that case) each time.
- Assume we have a list $a(1:n)$ which we need to sort. If we reduce it by one then that means we need to sort the smaller list $a(1:n-1)$.
- Assume for now that the smaller list $a(1:n-1)$ is sorted. Then to sort the original list $a(1:n)$ we just need to determine where $a(n)$ must be inserted in $a(1:n-1)$. There are several ways to do this.
- One way to do this is scan $a(1:n-1)$ from left to right and find the first element which is greater than or equal to $a(n)$; then we simply insert $a(n)$ before this element.
- Of course we can scan $a(1:n-1)$ from right to left and find the first element which is less than or equal to $a(n)$; then we simply insert $a(n)$ after this element. These are essentially equivalent but scanning from right to left is usually the one implemented. This is called *(straight) insertion sort*.

- We have already encountered another technique to search an array besides sequential search; remember that binary search was, in general, more efficient. If we use binary search to locate the position to insert $a(n)$ then the method is called *binary insertion sort*.
- Of course the algorithm is applied recursively as the following example demonstrates.



Straight Insertion Sort

Input: An array $a(1:n)$ of orderable elements

Output: An array $a(1:n)$ which is sorted in nondecreasing order

```
for i = 2:n
    v= a(i)
    j=i-1
    while j ≥ 0 and a(j)> v
      a(j+1) ←a(j)
      j=j-1
    end while
    a(j+1) ←v
end for
```


Example Apply the (straight) Insertion sort algorithm to sort the array

$$a = \{56, 43, 48, 22, 67, 29\}$$

Apply the algorithm recursively in a "bottom up" manner, i.e., by starting with an array of length one. Scan from right to left.

1. Start with the sorted array $\{56\}$ and we want to insert 43; we see that $56 > 43$ so we now have the sorted list $\{43, 56\}$.
2. We have the sorted list $\{43, 56\}$ and we want to insert 48; we scan to see that $56 > 48$ and $43 < 48$ so we add 48 before 56 to get $\{43, 48, 56\}$
3. We have the sorted list $\{43, 48, 56\}$ and we want to insert 22; we scan to see that the all elements including the first element are > 22 so we put 22 at the beginning to get $\{22, 43, 48, 56\}$
4. We have the sorted list $\{22, 43, 48, 56\}$ and we want to insert 67; we scan to see that element $56 < 67$ so we put 67 at the end to get $\{22, 43, 48, 56, 67\}$
5. We have the sorted list $\{22, 43, 48, 56, 67\}$ and we want to insert 29; we scan to see that all elements are > 29 and we reach the first $22 < 49$ so we put 29 after the first element to get the final sorted array $\{22, 29, 43, 48, 56, 67\}$

What is the complexity?

in worst case We have n^2 comparison and swamp.

Fake Coin Problem

There are several versions of this famous problem but the one we consider is that we are given n coins which look exactly alike but one is fake. For now assume the fake is slightly lighter than the real coins. The problem is to determine the fake coin using a balance.

Even if you didn't know about the Decrease and Conquer strategy, you would probably solve the problem using this approach:

- If n is even then we put half the coins on each side of the balance. The side which is lightest contains the fake coin.
- If n is odd, then **(n–1)** is even and we split the **(n–1)** coins in half and put each half on the balance. If both sides are equal weight, then we are done because the coin we left out is the fake one. If the balance is not even then we choose the lightest pile of coins to be the one containing the fake.
- We continue in this manner until we have found the fake coin by reducing the problem to weighing one on each side of the balance or found it by it being the one we didn't weigh.

Example Suppose we have 8 coins and we want to find which one is the fake coin; assume that we know the fake coin is lighter than the real ones. In how many steps can you guarantee to find the fake coin? What are the steps?

1. Put 4 coins on each side of the balance. Discard the coins on the side that is heavier because we know the fake coin is on the lighter side.
2. From the 4 put 2 coins on each side of the balance. Discard the coins on the side that is heavier. We now know that the fake coin is one of two.
3. Put one coin on each side. The coin that is lighter is the fake coin.

Example What is the difference in the strategy if we have 9 coins? Will it take more steps to do 9 coins?

We start by putting 4 coins on each side of the balance and keep one to the side. If the balance is level then the fake coin is the one to the side. If the balance is not level then we know the coin to the side is not fake but rather the fake is on the side of the balance that is lighter and we proceed as in the previous example. It should take no more than 3 steps.

Example Suppose we have 12 coins and we want to find which one is the fake coin; assume that we know the fake coin is lighter than the real ones. In how many steps can you guarantee to find the fake coin? What are the steps? How do the number of steps compare with the 8 coin example?

1. Put 6 coins on each side of the balance. Discard the coins on the side that is heavier because we know the fake coin is on the lighter side.
2. From the 6 put 3 coins on each side of the balance. Discard the coins on the side that is heavier. We now know that the fake coin is one of three.
3. Put one coin on each side and leave the other off the balance. If one of the coins on the balance is lighter, then it is the fake. If the balance is level then the coin to the side is the fake.

Note that this took the same number of steps as the 8 coins.

Example Suppose we have 8 coins and we want to find which one is the fake coin; assume that we do NOT know whether the fake coin is lighter or heavier than the real ones. How can we modify our algorithm to handle this case?

1. Put 4 coins on each side of the balance. For now set aside the 4 coins on the side that is heavier.
2. From the 4 lighter coins put 2 coins on each side of the balance.
   - If the balance is level we know that the fake coin is heavier and that it is one of the four coins we set aside. Thus we have to weigh the four heavier coins with two on each side to detect which is heavier.
   - If the balance is not level then we know the fake coin is lighter and we choose the 2 lighter coins
3. We now know that the take coin is one of two so we put one coin on each side. The coin that is lighter/heavier is the fake coin.

It may cost us one additional measurement to determine whether the fake coin is lighter or heavier so in general it will take 4 steps to decide which is the fake coin when we start with 8 coins.

Challenge yourself ☺

- http://www.primarygames.com/math/coinweighing/

Transform and Conquer Algorithms

A common approach to solving a problem is to transform it into one that is easier to solve. If the transformation costs are not prohibitive this can be an effective strategy.

In Gaussian elimination we transform a general linear matrix problem Ax = b into an equivalent one where the coefficient matrix is upper triangular which is much simpler to solve.

Checking element uniqueness in an array

Suppose we have an array of length n and we want to see if any two elements are equal. The brute force approach is to check all possible pairs; the worst case scenario for this is $\mathcal{O}(n^2)$ because we have to check $a(1)$ with $a(i)$, $i = 2, n$; then we check $a(2)$ with $a(i)$, $i = 3, n$. However, if we transform the array into a sorted array first, then all we have to do is check consecutive elements. Now the efficiency is determined by the work required for sorting and for the check of consecutive elements. The later is only $(n - 1)$ comparisons but the former depends on which sorting routine we choose. If we use Selection Sort or Bubble Sort these are $\mathcal{O}(n^2)$ and so the overall performance is $\mathcal{O}(n^2)$ which is the same as brute force. However, if we choose Mergesort then it is $\mathcal{O}(n \log n)$ and the overall result is $\mathcal{O}(n \log n)$ which is an improvement over brute force.

Searching in an array

Suppose we want to search an array of length n using a search key $\mathcal{K}$. The brute force approach is Sequential Search which just checks the n elements in the array so it is $\mathcal{O}(n)$. However, in the previous problem we found that sorting the list first improved the growth factor. If we sort the array first then the best we can do is $\mathcal{O}(n \log n)$ and if we use Binary Search then it is an additional $\mathcal{O}(\log n)$ so overall we have $\mathcal{O}(n \log n)$. So the result is worse! However, if we want to search an array multiple times with different keys it will pay to presort the array if you have enough searches.

Example:

Suppose we have an array of length 1000 which we want to search m times. If m = 1 then it is not efficient to first sort the array but if m is large, then it is better to sort first. Approximately how large should m be so that it is more efficient to sort first?

If we do m sequential searches of a non-sorted list then the work is approximately m(n). Sometimes this is written as m(n/2) because on average we will find the key by the time we have searched half way through the array. However, the ½ is just a constant and won't affect the power of n so we omit it here.

If we sort the list first by Mergesort then that requires **O**(n log n). Then to perform m searches of a sorted array of length n using Binary Search requires m**O**(log n).

Comparing these we determine when

$$mn = n \log n + m \log n = (n + m) \log n$$

for our choice of $n = 1000$. We have

$$1000m = (1000 + m)(6.9) = 6900 + 6.9m \implies 993m \approx 6900$$

where we have chosen base e, i.e., $\ln n$. This says that if we do 7 searches it is

probably better to sort first. (6900/993=6.95)