CSC 225 - Summer 2019

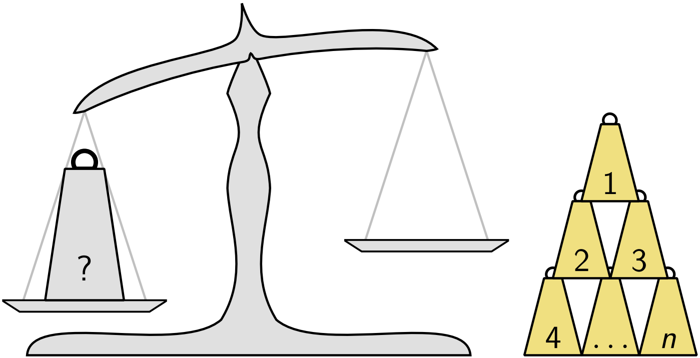
Algorithm Anaylsis IV

Bill Bird

Department of Computer Science
University of Victoria

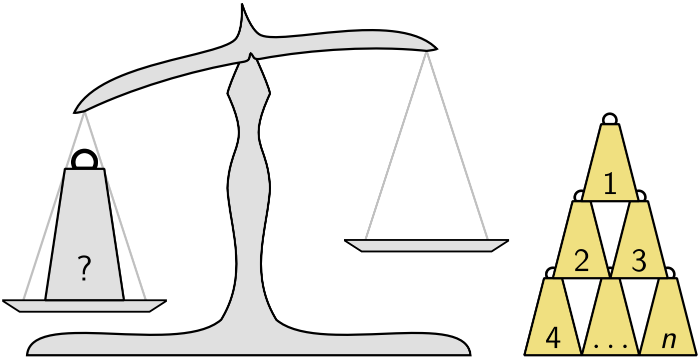
May 17, 2019

A Balancing Problem (1)



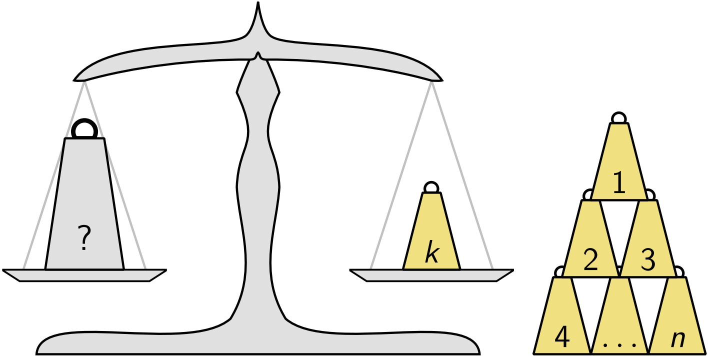
- The unknown weight (in grey) on the scale above has an integer weight between 1 and n units.
- **Problem**: Given labelled weights with values $1, 2, \ldots, n$, find a weight which balances the scale.

A Balancing Problem (2)



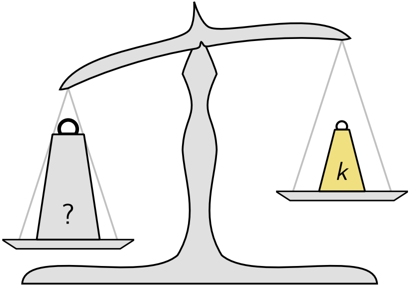
- Before we can create an algorithm for this problem, we need to formalize it in a mathematical way.

A Balancing Problem (3)

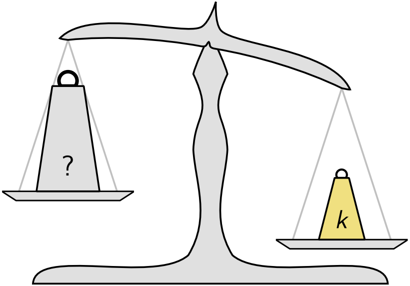


- First, we should note the restrictions:
  - We want to balance the scale using **one** weight.
  - The only way to test a weight is by putting it on the scale.

A Balancing Problem (4)



$\textsc{Weigh}(k) =$ TOO_LIGHT



$\textsc{Weigh}(k) =$ TOO_HEAVY

- We can model putting a weight with value k on the scale with a function $\textsc{Weigh}$.
- For a weight k, $\textsc{Weigh}(k)$ returns either TOO_LIGHT, BALANCED or TOO_HEAVY.

A Balancing Problem (5)

$\textsc{OneWeightBalance}$

Input: An integer n and a function $\textsc{Weigh}$.

Output: An integer $k \in \{1, 2, \ldots, n\}$ such that

$$\textsc{Weigh}(k) = \text{BALANCED}.$$

Linear-time solution (1)

procedure $\textsc{BalanceLinear}(n)$
  for $k = 1, 2, 3, \ldots, n$ **do**
    if $\textsc{Weigh}(k) =$ BALANCED **then**
      return k
    end if
  end for
end procedure

- Simple solution: Try each weight from 1 to n. Since the value of the grey weight must be an integer between 1 and n, a solution is guaranteed to exist.
- Assuming that the $\textsc{Weigh}$ function is constant-time, this solution is $\Theta(n)$.

Aside: Names

Common asymptotic complexity classes are often referred to by their English names.

Symbolic Name	**English Name**
$\Theta(1)$	Constant
$\Theta(\log n)$	Logarithmic
$\Theta(n)$	Linear
$\Theta(n \log n)$	Linearithmic[1]
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^c)$	Polynomial (when c is constant)
$\Theta(2^n)$	Exponential

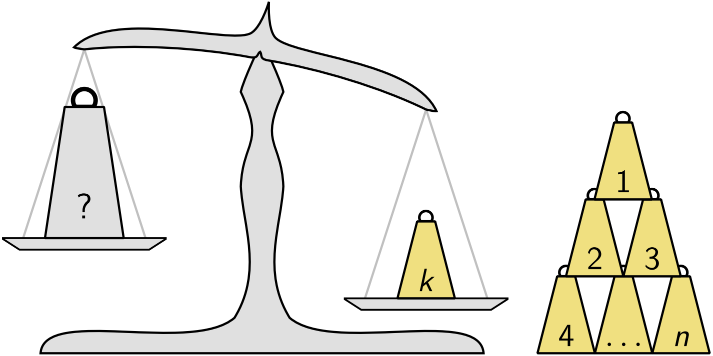
[1] Often better to just say 'n log n'

Linear-time solution (1)

```
procedure BalanceLinear(n)
    for k = 1, 2, 3, ..., n do
        if Weigh(k) = BALANCED then
            return k
        end if
    end for
end procedure
```

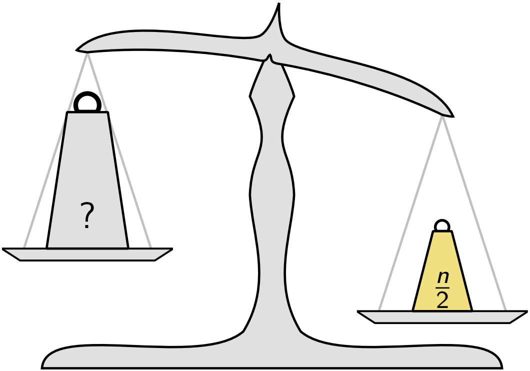
- Can we do better than $\Theta(n)$ for this problem?
- If not, can we prove that no faster algorithm exists?

Improved Solution (1)



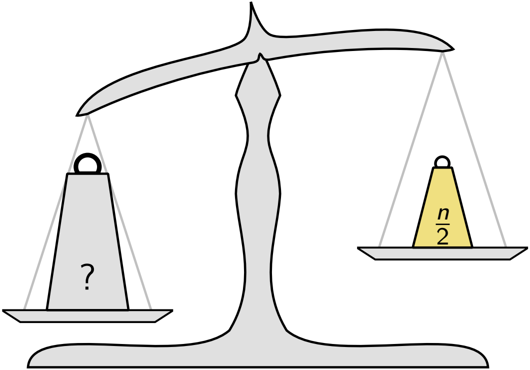
- **Observation**: If we weigh k and find that it's too heavy, we can ignore all weights greater than k.

Improved Solution (2)



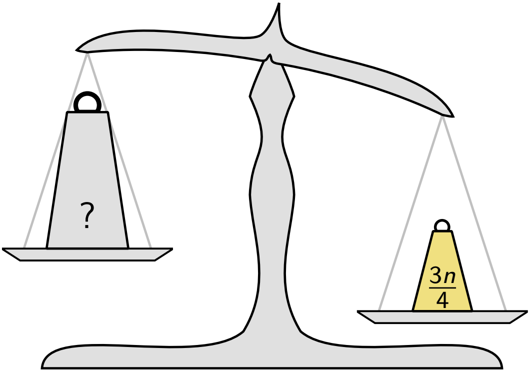
- **Idea**: Keep track of the highest and lowest possible values for the unknown weight, and narrow down the possibilities by weighing the midpoint of the two.
- **Spoiler**: The algorithm we're going to create is binary search.

Improved Solution (3)



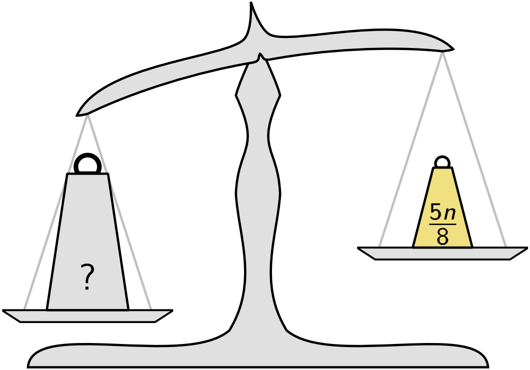
- At first, the lowest possible weight is 1 and the highest is n.
- The midpoint of 1 and n is $n/2$.
- If $n/2$ is too light, then it becomes the new lowest possibility.

Improved Solution (4)



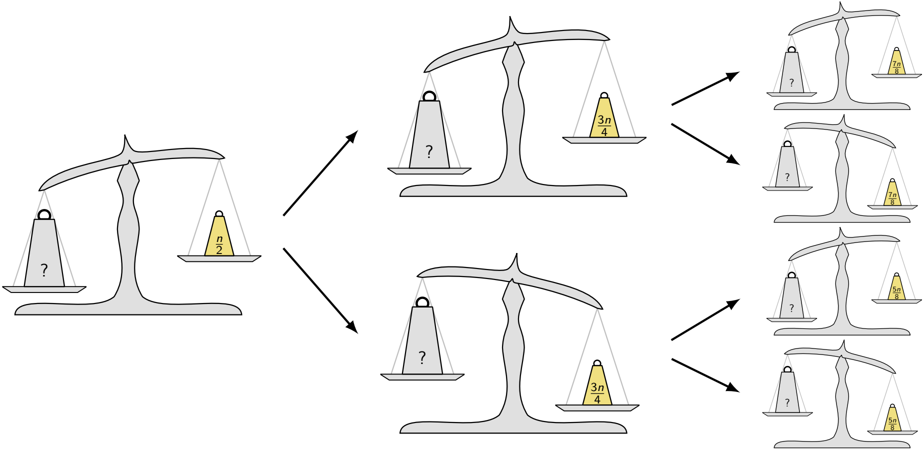
- The midpoint of $n/2$ and n is $3n/4$.
- If $3n/4$ is too heavy, it becomes the new highest possibility.

Improved Solution (5)



- The midpoint of $n/2$ and $3n/4$ is $5n/8$.
- Eventually, this process will converge to the actual weight.

Improved Solution (6)

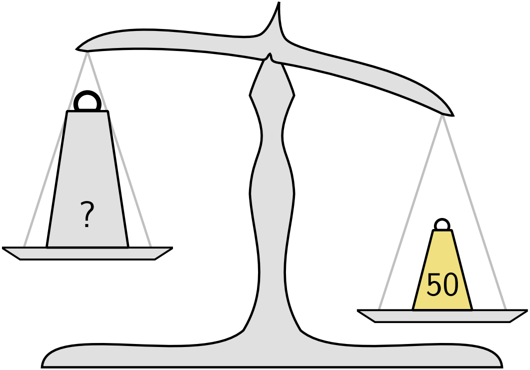


- By narrowing the range down using the scale's results, we can avoid testing many of the possible weights.

Improved Solution (7)

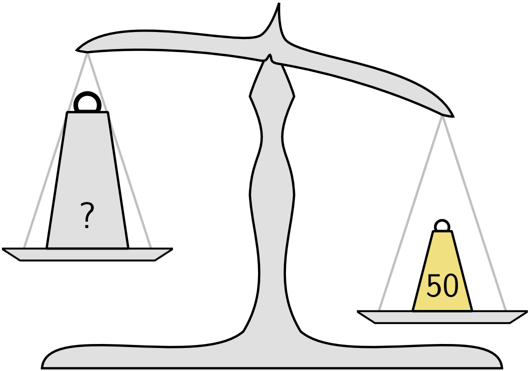
```
procedure BalanceImproved(n)
    high ← n
    low ← 1
    while low < high do
        k ← (high + low)/2
        if Weigh(k) = TOO_HEAVY then
            high ← k − 1
        else if Weigh(k) = TOO_LIGHT then
            low ← k + 1
        else
            //The weight must be equal to k
            return k
        end if
    end while
    //At this point, low = high
    return low
end procedure
```

Example of Improved Algorithm (1)



- **Example**: $n = 100$.

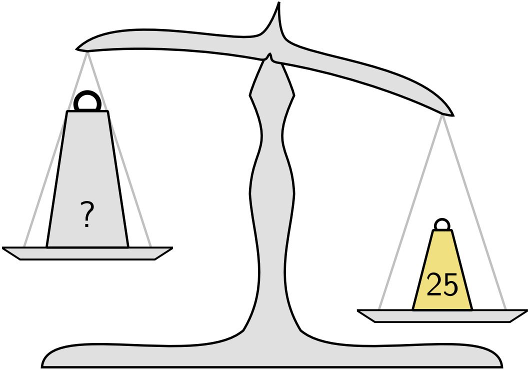
Example of Improved Algorithm (2)



low	k	high
1	50	100

$\textsc{Weigh}(50) =$ TOO_HEAVY, so set `high` $= 49$.

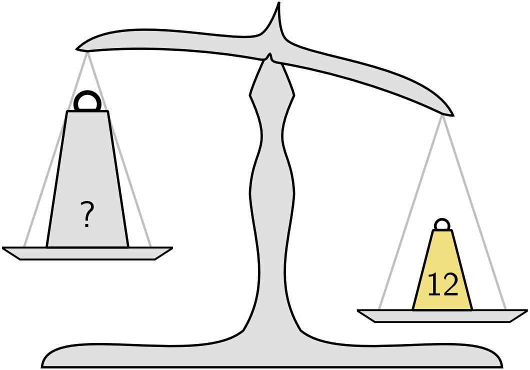
Example of Improved Algorithm (3)



low	k	high
1	25	49

$\textsc{Weigh}(25) =$ TOO_HEAVY, so set `high` $= 24$.

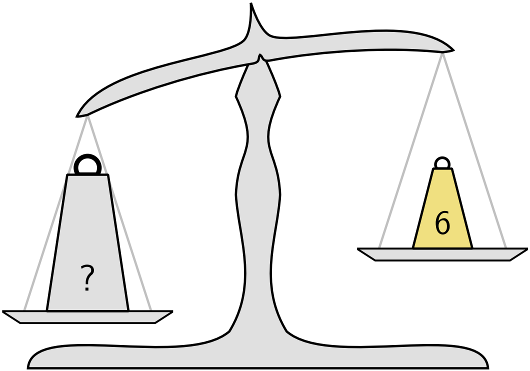
Example of Improved Algorithm (4)



low	k	high
1	12	24

$\textsc{Weigh}(12) =$ TOO_HEAVY, so set `high` $= 11$.

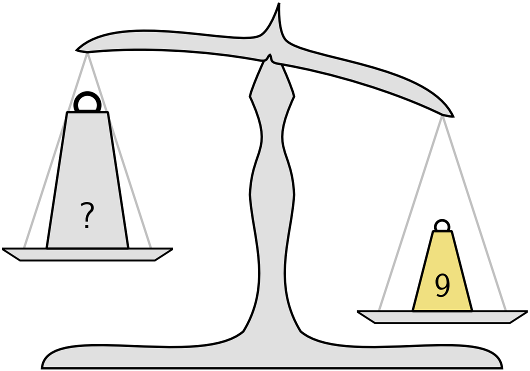
Example of Improved Algorithm (5)



low	k	high
1	6	11

$\text{WEIGH}(6) = \text{TOO\_LIGHT}$, so set low $= 7$.

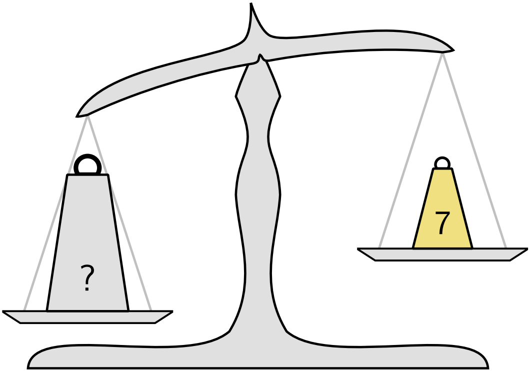
Example of Improved Algorithm (6)



low	k	high
7	9	11

$\textsc{Weigh}(9) =$ TOO_HEAVY, so set `high` $= 8$.

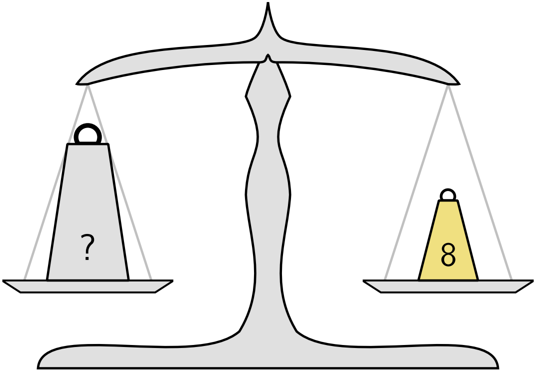
Example of Improved Algorithm (7)



low	k	high
7	7	8

$\textsc{Weigh}(7) = \text{TOO\_LIGHT}$, so set `low` $= 8$.

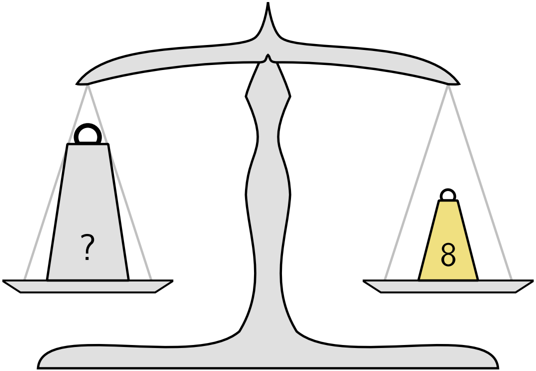
Example of Improved Algorithm (8)



low	k	high
8	8	8

low = high, so the correct weight must be 8.

Example of Improved Algorithm (9)



- $n = 100$
- The improved algorithm found the correct value after 7 weighings.
- What is the worst-case running time of the improved algorithm?

Analysis of Improved Algorithm (1)

```
procedure BALANCEIMPROVED(n)
    high ← n
    low ← 1
    while low < high do
        k ← (high + low)/2
        if WEIGH(k) = TOO_HEAVY then
            high ← k − 1
        else if WEIGH(k) = TOO_LIGHT then
            low ← k + 1
        else
            //The weight must be equal to k
            return k
        end if
    end while
    //At this point, low = high
    return low
end procedure
```

The code inside the loop requires constant time.

Analysis of Improved Algorithm (2)

```
procedure BalanceImproved(n)
    high ← n
    low ← 1
    while low < high do
        k ← (high + low)/2
        if Weigh(k) = TOO_HEAVY then
            high ← k − 1
        else if Weigh(k) = TOO_LIGHT then
            low ← k + 1
        else
            //The weight must be equal to k
            return k
        end if
    end while
    //At this point, low = high
    return low
end procedure
```

How many iterations does the loop require in the worst case?

Analysis of Improved Algorithm (3)

The loop terminates when $\texttt{low} = \texttt{high}$ (that is, when the range is narrowed down to a single value).

At each step, the number of values in the range is

$$\texttt{high} - \texttt{low} + 1.$$

At each iteration, either

$$\texttt{low} \leftarrow (\texttt{high} + \texttt{low})/2 + 1$$

or

$$\texttt{high} \leftarrow (\texttt{high} + \texttt{low})/2 - 1$$

Analysis of Improved Algorithm (4)

Claim: At each iteration, the size of the range is halved.

If the new lower bound is

$$(\texttt{high} - \texttt{low})/2 + 1,$$

then the new size is

$$\texttt{high} - [(\texttt{high} - \texttt{low})/2 + 1] + 1 = \frac{1}{2}(\texttt{high} - \texttt{low} + 1)$$

If the new upper bound is

$$(\texttt{high} - \texttt{low})/2 - 1,$$

then the new size is

$$[(\texttt{high} - \texttt{low})/2 - 1] - \texttt{low} + 1 = \frac{1}{2}(\texttt{high} - \texttt{low} + 1)$$

Analysis of Improved Algorithm (5)

Iteration	**Range Size**
0	n
1	$\frac{n}{2}$
2	$\frac{n}{2^2}$
3	$\frac{n}{2^3}$
4	$\frac{n}{2^4}$
$\vdots$	$\vdots$
k	$\frac{n}{2^k}$

Analysis of Improved Algorithm (6)

The algorithm terminates after k iterations, where

$$\frac{n}{2^k} = 1$$

The total number of iterations is then

$$k = \log_2 n$$

Therefore, the improved algorithm requires $\Theta(\log n)$ operations.

Searching in Arrays

	ArraySearch
Input:	An array A of n integers and an integer k.
Output:	An index i such that $A[i] = k$, or -1 if no such index exists.

	SortedArraySearch
Input:	A sorted array A of n integers and an integer k.
Output:	An index i such that $A[i] = k$, or -1 if no such index exists.

Linear Search

procedure LINEARSEARCH(A, n, k)
    for $i = 0, 1, 2, \ldots, n-1$ **do**
        if $A[i] = k$ **then**
            return i
        end if
    end for
    return -1
end procedure

- The LINEARSEARCH function above is a $\Theta(n)$ solution to both ARRAYSEARCH and SORTEDARRAYSEARCH.

Binary Search

```
procedure BinarySearch(A, n, k)
    high ← n − 1
    low ← 0
    while low < high do
        i ← (high + low)/2
        if A[i] > k then
            high ← i − 1
        else if A[i] < k then
            low ← k + 1
        else
            return i
        end if
    end while
    if A[low] = k then
        return low
    else
        return −1
    end if
end procedure
```

- BinarySearch is a $\Theta(\log n)$ solution to the SortedArraySearch problem.

Unsorted Arrays (1)

ArraySearch

Input:	An array A of n integers and an integer k.
Output:	An index i such that $A[i] = k$, or -1 if no such index exists.

- When the array A is not sorted, is there an algorithm which is faster than $\Theta(n)$?

Unsorted Arrays (2)

Theorem: Every algorithm to solve the ArraySearch problem requires

$$\Omega(n)$$

operations in the worst case.

- The Theorem above can be proven by showing that since the array is not sorted, any algorithm must inspect all n elements to determine whether k is in the array.
- We can say that $\Omega(n)$ is a *lower bound* for searching an unsorted array.
- An algorithm that achieves the lower bound is said to be **asymptotically optimal**.

Unsorted Arrays (3)

Theorem: Every algorithm to solve the ArraySearch problem requires

$$\Omega(n)$$

operations in the worst case.

Proof:
We will use a proof by contradiction to show that every algorithm must inspect all n elements of the array A.

Suppose there was an algorithm that solved ArraySearch without inspecting every element of A.

Consider the array A shown below.

A[0]	A[1]	A[2]	...	A[j]	...	A[$n-1$]
0	1	2	...	j	...	n

Unsorted Arrays (4)

Proof:

Consider the array A shown below.

A[0]	A[1]	A[2]	...	A[j]	...	A[$n-1$]
0	1	2	...	j	...	n

Since the algorithm does not inspect every element of A, there must be some element $A[j]$ which the algorithm does not examine.

Since $A[j] = j$, and j is not present anywhere else in the array, the return value of $\textsc{ArraySearch}(A, n, j)$ must be j.

But since the algorithm does not inspect index j, if we set $A[j] = j + 1$, the return value of $\textsc{ArraySearch}(A, n, j)$ must still be j, even though j will no longer be in the array. Therefore, the algorithm cannot be correct, which is a contradiction.

Analysis with Multiple Parameters (1)

TestIntersection

Input: An array A of n integers and an array B of m integers.
Output: `true` if there is any element that appears in both A and B. `false` otherwise.

Analysis with Multiple Parameters (2)

```
procedure TestIntersection(A, n, B, m)
    for i = 0, 1, 2, ..., n − 1 do
        for k = 0, 1, 2, ..., m − 1 do
            if A[i] = B[k] then
                return true
            end if
        end for
    end for
    return false
end procedure
```

The pseudocode above gives one possible algorithm for the TestIntersection problem.

Analysis with Multiple Parameters (3)

procedure TESTINTERSECTION(A, n, B, m)
    for $i = 0, 1, 2, \ldots, n-1$ **do**
        for $k = 0, 1, 2, \ldots, m-1$ **do**
            if $A[i] = B[k]$ **then**
                return `true`
            end if
        end for
    end for
    return `false`
end procedure

Question: How can we describe the running time of this algorithm?

Analysis with Multiple Parameters (4)

```
procedure TestIntersection(A, n, B, m)
    for i = 0, 1, 2, ..., n − 1 do
        for k = 0, 1, 2, ..., m − 1 do
            if A[i] = B[k] then
                return true
            end if
        end for
    end for
    return false
end procedure
```

The algorithm depends on both the size of A (given by n) and the size of B (given by m), and there is no direct relationship between n and m. Therefore, both n and m should appear in any expression of the running time.

Analysis with Multiple Parameters (5)

```
procedure TestIntersection(A, n, B, m)
    for i = 0, 1, 2, ..., n − 1 do
        for k = 0, 1, 2, ..., m − 1 do
            if A[i] = B[k] then
                return true
            end if
        end for
    end for
    return false
end procedure
```

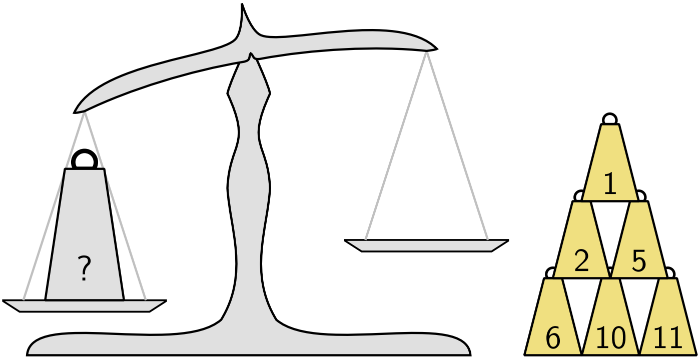
This algorithm is $\Theta(nm)$ in the worst case (and $\Theta(1)$ in the best case).

Analysis with Multiple Parameters (6)

```
procedure MaxElement(A, n, B, m)
    max ← A[0]
    for i = 1, 2, ..., n − 1 do
        if max < A[i] then
            max ← A[i]
        end if
    end for
    for i = 0, 1, 2, ..., m − 1 do
        if max < B[i] then
            max ← B[i]
        end if
    end for
    return max
end procedure
```

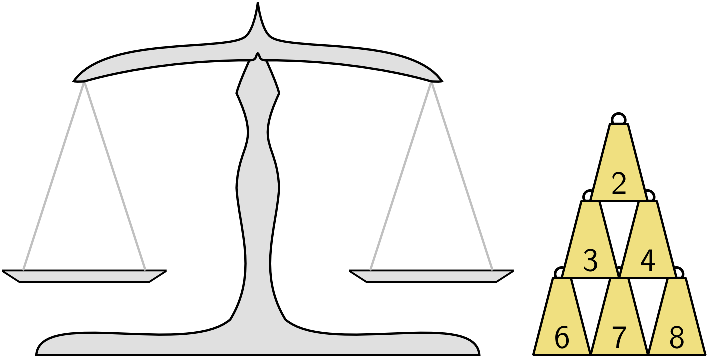
Similarly, the above algorithm (which finds the maximum element across both of A and B) has running time $\Theta(n + m)$.

More Balancing Problems (1)



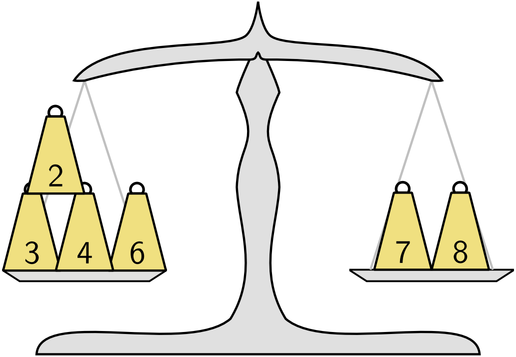
- **Problem**: Given an unbalanced scale and a collection of **arbitrary** weights, balance the scale.

More Balancing Problems (2)



- **Problem**: Given an empty scale and a collection of arbitrary weights, find a way to arrange *all* weights such that the scale is balanced.

More Balancing Problems (3)



- **Problem**: Given an empty scale and a collection of arbitrary weights, find a way to arrange *all* weights such that the scale is balanced.

Balancing Problems (1)

OneWeightBalance

- Given weights with values $1, 2, \ldots, n$, find the value of an unknown weight with value between 1 and n.
- Use only **one** weight; a solution always exists.
- Best known algorithm: $\Theta(\log n)$ (Binary Search)
- The $\Theta(\log n)$ algorithm is optimal (proven in CSC 226).

PowersOfTwoBalance

- Given weights with values $2^0, 2^1, \ldots, 2^n$, find the value of an unknown weight with value between 1 and $2^{n+1} - 1$.
- Multiple weights may be needed; a solution always exists.
- Best known algorithm: $\Theta(n)$ (Greedy Heuristic/Binary Search)
- The $\Theta(n)$ algorithm is optimal.

Balancing Problems (2)

ArbitraryBalance

- Given an arbitrary collection of weights, find the value of an unknown weight.
- Multiple weights may be needed; solution may not exist.
- Best known algorithm: $O(2^n)$ (Exhaustive Search).
- No polynomial time algorithm is known.
- The existence of a polynomial time algorithm depends on the P vs. NP problem (see CSC 226 and CSC 320).

Partition

- Given an arbitrary collection of weights and an empty scale, find an arrangement of weights which balances the scale.
- Solution may not exist.
- Best known algorithm: $O(2^n)$ (Exhaustive Search)
- The existence of a polynomial time algorithm also depends on the P vs. NP problem.