Notes

Analysis of Algorithms, Asymptotic Notations

Algorithms

several definitions To name a few:

- a comprehensive list of actions that must be completed in order.
- a set of guidelines that properly describes an action flow.
- a series of actions to be taken in order to address an issue.
- A well-defined series of steps used to solve a well-defined problem in a finite amount of time is known as an algorithm.

What is an Algorithm

An algorithm is a collection of instructions that must be followed step by step in order to solve a problem.

A computer's potential solution to a problem is frequently described using algorithms. An algorithm can take the form of a recipe.

It explains the steps to take and the ingredients needed to prepare the dish. If the recipe provides clear instructions without any ambiguity, it is an algorithm.

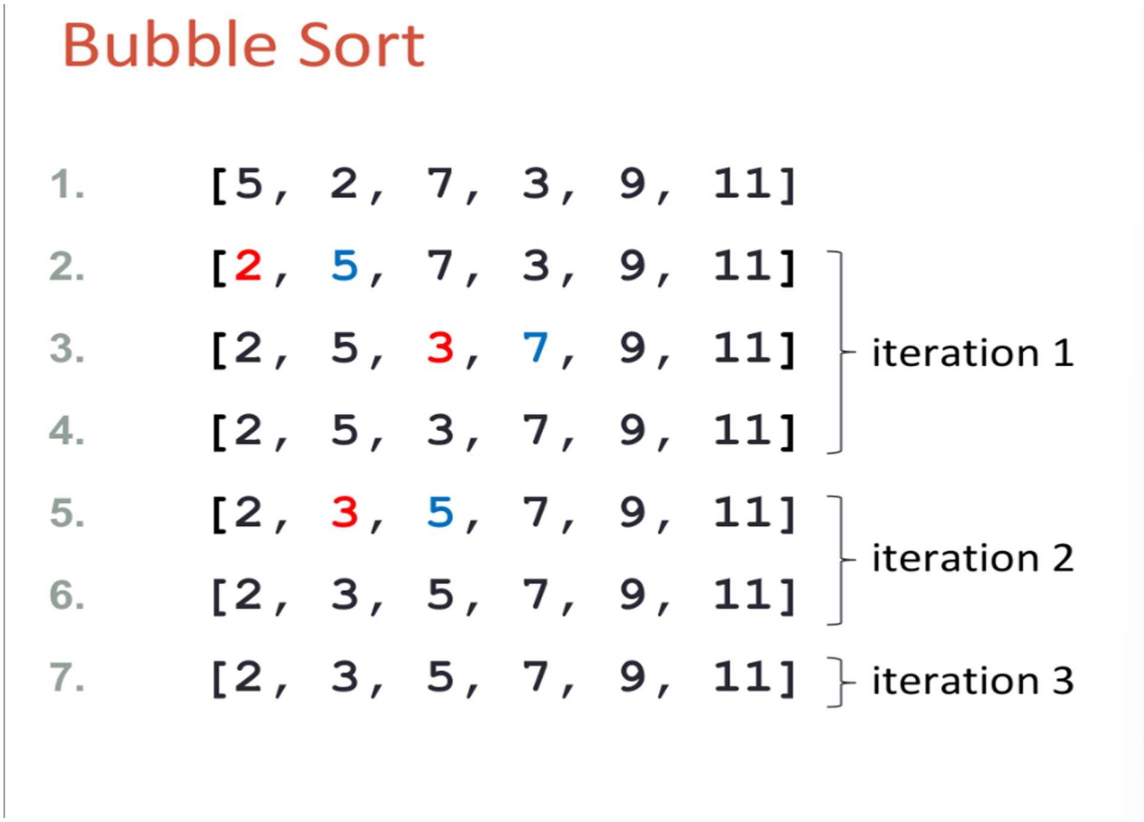
How does this algorithm work?

Given is an array A of size n of integer numbers.

```
for ( i = 1 ; i < n ; i++ )
{
	int swaps = 0;
	for ( j = 0 ; j < n – i; j++ )
	{
		if ( a[j] > a[j + 1] )
		{
			swap( a[j], a[j + 1] );
			swaps = swaps + 1;
		}
	}
	if ( swaps == 0 )  then  break;
}
```

Bubble sort


```
for ( i = 1 ; i < n ; i++ )
{
	int swaps = 0;
	for ( j = 0 ; j < n – i; j++ )
	{
		if ( a[j] > a[j + 1] )
		{
			swap( a[j], a[j + 1] );
			swaps = swaps + 1;
		}
	}
	if ( swaps == 0 )  then  break;
}
```



Bubble Sort

It is a useless sorting algorithm because others with similar complexity typically operate more quickly.

It rapidly recognizes that the list is sorted, which is a positive thing. It completes this task more quickly than some alternative (better) algorithms.

It is not the only one doing this, though.

How does this algorithm work?

Given is an array A of size n of integer numbers.

```
for ( i = 1 ; i < n ; i++ )
{
        j = i;
        while ( ( j > 0 ) && ( A[j-1] > A[j] ) )
        {
                swap( A[j], A[j-1] );
                j = j – 1;
        }
}
```

Insertion Sort

Given is an array A of size n of integer numbers.

```
for ( i = 1 ; i < n ; i++ )
{
      j = i;
    j=
    A[1] ==45, a[j-i] = a[0] =23
    23 45 78------
      while ( ( j > 0 ) && ( A[j-1] > A[j] ) )
      {
            swap( A[j], A[j-1] );
            j = j – 1;
      }
}
```

Insertion Sort

1. [5, 2, 2, 3, 9, 11]
2. [5, 2, 2, 3, 9, 11]
3. [5, 2, 2, 3, 9, 11]
4. [2, 5, 2, 3, 9, 11]
5. [2, 5, 2, 3, 9, 11]
6. [2, 2, 5, 3, 9, 11]
7. [2, 5, 3, 7, 9, 11]
8. [2, 3, 5, 7, 9, 11]
9. [2, 3, 5, 7, 9, 11]
10. [2, 3, 5, 7, 9, 11]
11. [2, 3, 5, 7, 9, 11]

14,1,23, 8,7,11,2,5

- 14,1,23,8,7,11,2,5
- 1,14,23,8,7,11,2,5
- 1,14, 23,8,7,11,2,5
- 1,8,14,23,7,11,2,5
- 1,7,8,14,23,11,2,5
- 1,7,8,11,14,23,2,5
- 1,2,7,8,11,14,23,5
- 1,2,5,7,8,11,14,23

Insertion Sort

Adaptive, Insertion Sort will function much more efficiently if it is applied to a nearly sorted list. There is a variation in how the data is displayed!

The relative order of elements with equal keys does not change, making the system stable.

In-place, i.e., only needs a fixed quantity of extra memory elements

How does this algorithm work?

Given is an array A of size n of integer numbers.

```
for ( int i = 0 ; i < n; i++ )
{
    int min = i;
    for ( int j = i+1; j < n; j++ )
    {
        if ( A[j] < A[min] )  then  min = j;
    }
    if ( min != i )  then  swap( A[i], A[min] );
}
```

Selection Sort

Insertion Sort

1. [5, 2, 2, 3, 9, 11]
2. [5, 2, 2, 3, 9, 11]
3. [5, 2, 2, 3, 9, 11]
4. [2, 5, 2, 3, 9, 11]
5. [2, 5, 2, 3, 9, 11]
6. [2, 2, 5, 3, 9, 11]
7. [2, 5, 3, 7, 9, 11]
8. [2, 3, 5, 7, 9, 11]
9. [2, 3, 5, 7, 9, 11]
10. [2, 3, 5, 7, 9, 11]
11. [2, 3, 5, 7, 9, 11]

Selection Sort

In general, Selection Sort performs the same amount of comparisons, but unlike Insertion Sort, there is no "drop out."

This greatly reduces its effectiveness.

Experimental study of algorithms: Issues

Timing does not really evaluate the algorithm, but merely evaluates a specific implementation

- Use asymptotic analysis to evaluate an algorithm

1. Examine the algorithm itself, not the implementation
2. reason about performance as afunction of n
3. Methematically prove things about performance

- Use timing to evaluate an implementation

1. In the real world, we do want to know whether implementation A runs faster than implementation B on data set C

Running time T(n)

The running time of a given algorithm is the number of elementary operation to reach a solution , For example

- Sorting- number of array elements
- Arithmetic operation – number of bits
- Grraph search – number of vertices and edges

We can evaluate running time by

- Operation Counting
- Asymtotic Notations
- Substitution Method
- Recurrence Tree
- Master Method
- Operation Counting

```
int LinearSearch( int[] A, int n, int value )
{
      for ( int i = 0 ; i < n; i++ )     1 (initialization) + n+1 (checks) operations
      {
                  if ( A[i] == value ) n (checks) operations
                        then return i;       1 (return) operation
      }           n (increment) operations
      return -1;      1 (return) operation
}
```

- Worst case: $T(n) = 1 + (n+1) + n + 1 + n = 3n + 3$ operations
- Best case: $T(n) = 1 + 1 + 1 + 1 = 4$ operations

Best, Worst and Average Case Complexity

Running the same algorithm on different inputs may yield different running times.

Back to the Linear search algorithm

Worst case running time: What if the value not belong to the data array?

Best case running time: What if the first element matches the values to search for ?

Average case running time: what if the value exists some where in the middle of the data array?

Focus on Worst Case

When comparing two algorithms, consider the worst – case running time complexity. Why?

- It provides anupper bound on the runtimes
- It happens fairly often
- Average case usually is similar to the worst case.

Operation Counting: Dealing with nested loops

Code	Operations
InsertionSort(int[] A, int n)	
for (int j = 1 ; j < n; j++)	1 (initialization) + n (checks) operations
{	
int temp = A[j];	n-1 (assignment) operation
int i = j −1;	n-1 (assignment) operation
while ((i ≥ 0) && (A[i] > temp))	$2\sum_{j=1}^{n} j$-1 (checks) operations
{	
A[i + 1] = A[i];	$\sum_{j=1}^{n-1} j$ (assignment) operations
i = i−1;	$\sum_{j=1}^{n-1} j$ (decrement) operations
}	
A[i + 1] = temp;	n-1 (assignment) operations
}	n-1 (increment) operations

Worst case: $T(n) = 1 + n + (n-1) + (n-1) + 2(n(n+1)/2 - 1) + 2n(n-1)/2 + (n-1) + (n-1)$

$= 2n^2 + 5n - 5$ operations

Arithmetic series: $\sum_{j=1}^{n} j = \frac{1}{2}n(n+1)$

Operation Counting: General rules

Sequence: Add the running times of consecutive statements.

Loops: the running time of the inner loop statements times the product of the sizes of all loops

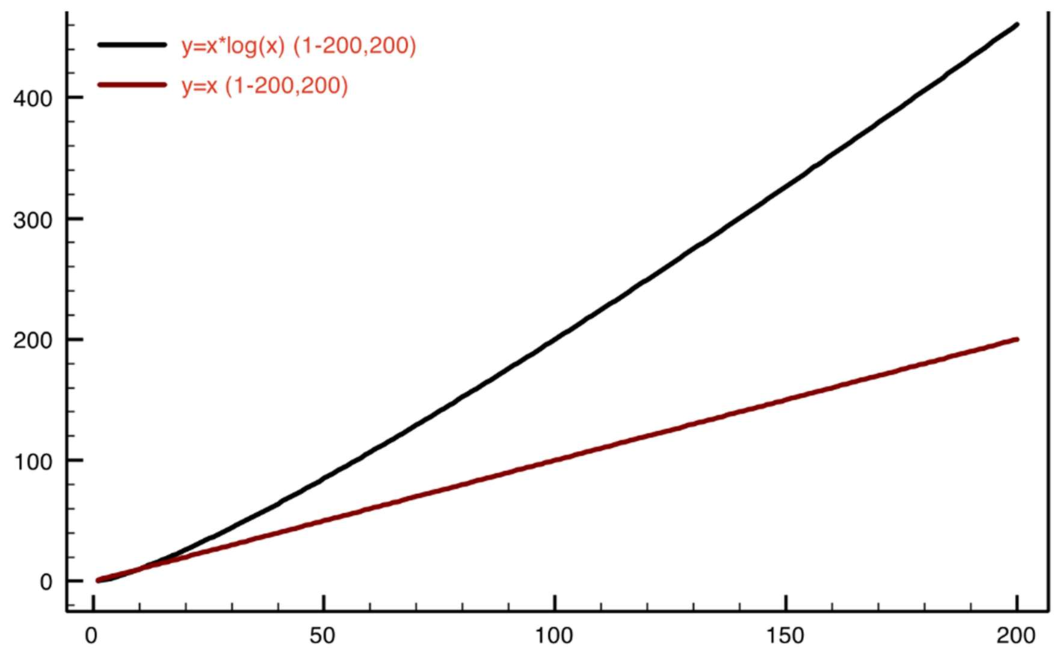
Control statement : the maximum of running times of case1, case2, …case k.

Asymptotic Notations

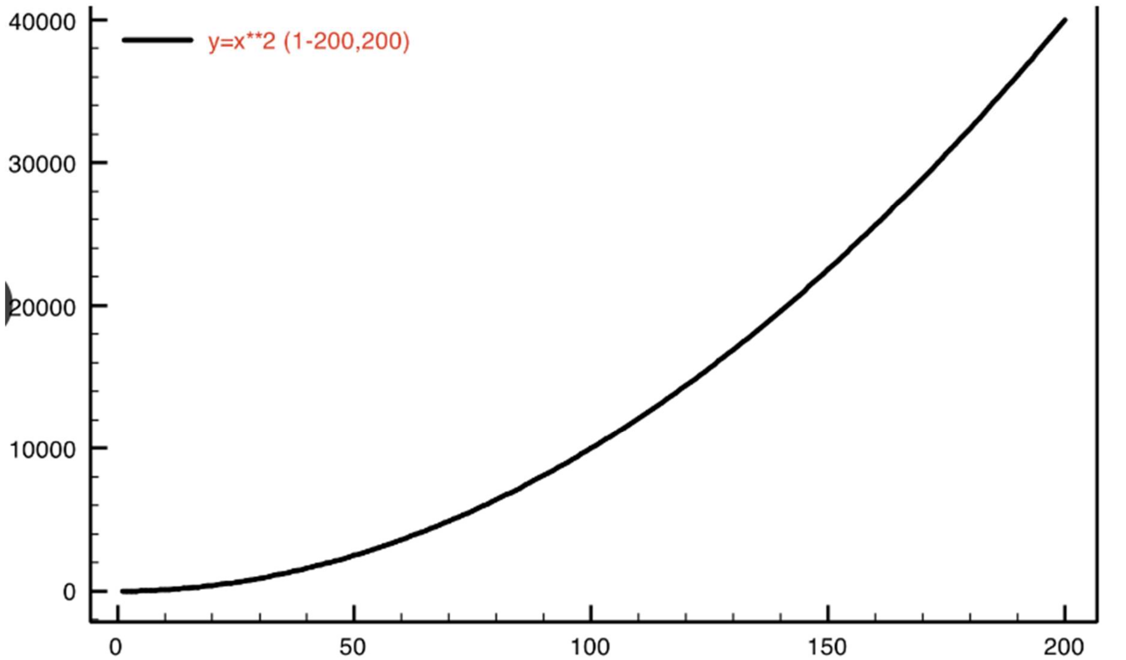
Efficiency: What’s our measure?

- Let n be the number of input elements

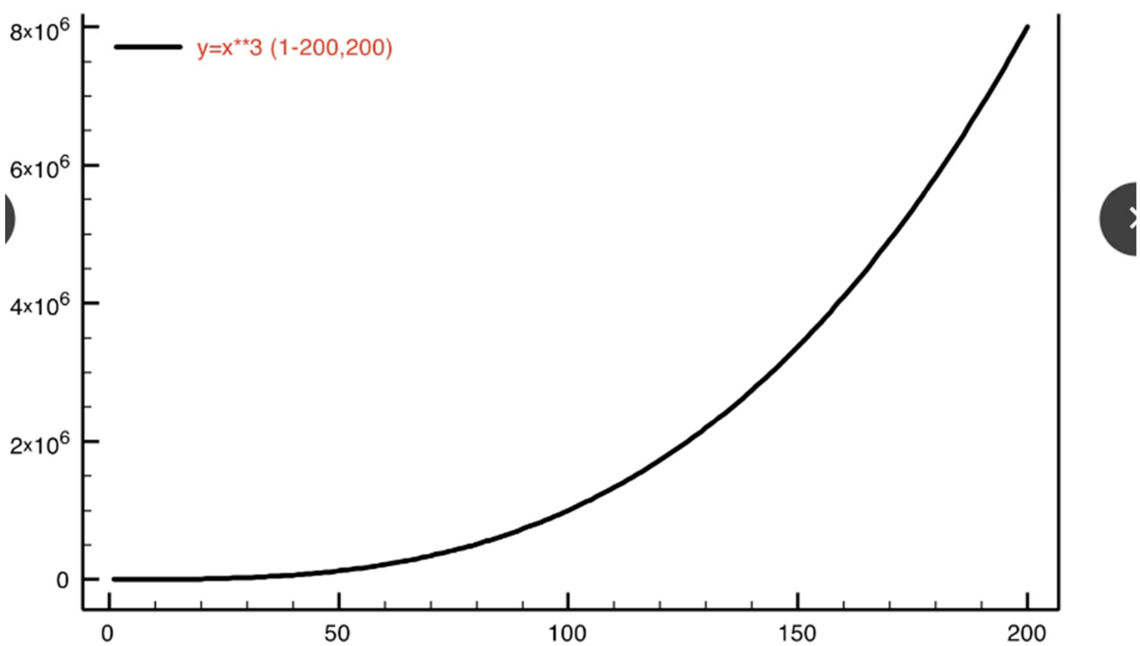
- (For example , the number of integers in the array).
- Measure time for operations on data structures and time to execute algorithms in dependence (function) of n.
- Consisder order of magnitude.
- Linear Runtime



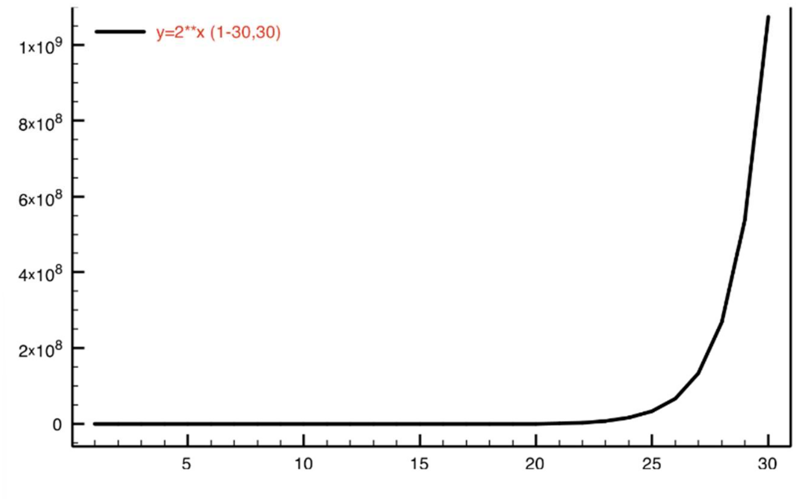
Quadratic Runtime



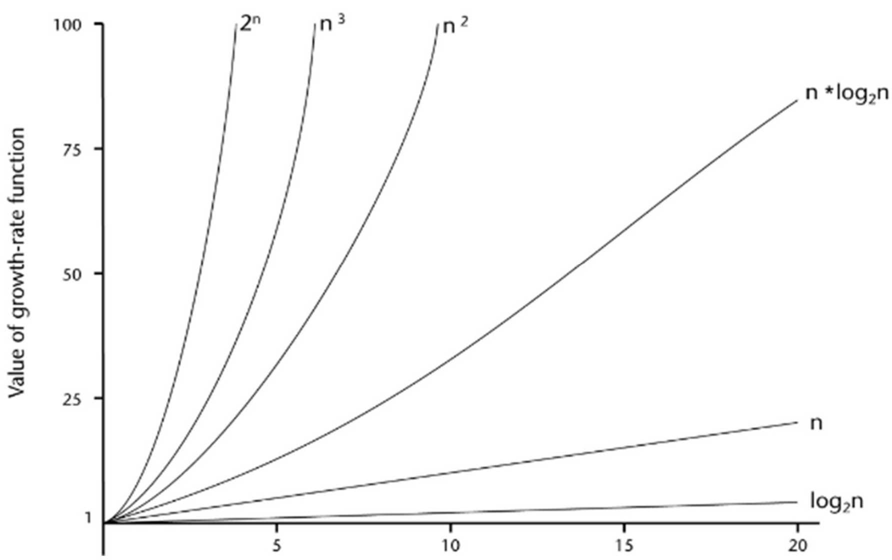
Cubic Runtime



Exponential Runtime



Order of growth



- Lower- order terms are not significant for large n
- Constants and coefficients are less significant

Complexity of Algorithms

Linear Search: Complexity

```
int LinearSearch( int[] A, int n, int value )
{
      for ( int i = 0 ; i < n; i++ )
      {
            if ( A[i] == value )
                  then return i;
      }
      return -1;
}
```

- Worst case: T(n) = $O(n)$ comparisons
- Best case: T(n) = $O(1)$ comparisons
- Average case: T(n) = $O(n)$ comparisons
- Worst-case space complexity: $O(1)$ iterative
- Overall algorithm: T(n) = $O(n)$ or $\Omega(1)$ we can not use θ as they are different

Linear Search: Average Case Analysis

- We shall focus on the probable position for the desired element x, rather than the elements of the array A.
- Since for any input A, the element x is equally likely to be present in any location of A. Therefore, $\Pr[A[i] = x] = \frac{1}{n}$ for $1 \leq i \leq n$.
- The cost of accessing the i^{th} location is i and the associated probability is $\frac{1}{n}$. Therefore, the expected cost is

$$1 \times \frac{1}{n} + 2 \times \frac{1}{n} + \cdots + n \times \frac{1}{n} = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}.$$

- Finally, $\frac{n+1}{2} = O(n)$ comparisons on average.

Bubble Sort: Complexity :

Given is an array A of size n of integer numbers.

```
for ( i = 1 ; i < n ; i++ )
{
    int swaps = 0;
    for ( j = 0 ; j < n – i; j++ )
    {
        if ( a[j] > a[j + 1] )
        {
            swap( a[j], a[j + 1] );
            swaps = swaps + 1;
        }
    }
    if ( swaps == 0 )  then  break;
}
```

- Worst case: T(n) = $O(n^2)$ comparisons
- Best case: T(n) = $O(n)$ comparisons
- Average case: T(n) = $O(n^2)$ comparisons
- Worst-case space complexity: $O(1)$ auxiliary
- Overall algorithm: T(n) = $O(n^2)$ or $\Omega(n)$ we can not use θ

Insertion Sort : Complexity

Given is an array A of size n of integer numbers.

```
for ( i = 1 ; i < n ; i++ )
{
    j = i;
    while ( ( j>0 ) && ( A[j-1] > A[j] ) )
    {
        swap( A[j], A[j-1] );
        j = j – 1;
    }
}
```

- Worst case: T(n) = $O(n^2)$ comparisons
- Best case: T(n) = $O(n)$ comparisons
- Average case: T(n) = $O(n^2)$ comparisons
- Worst-case space complexity: $O(1)$ auxiliary
- Overall algorithm: T(n) = $O(n^2)$ or $\Omega(n)$ we can not use θ

Selection Sort : Complexity

Given is an array A of size n of integer numbers.

```
for ( int i = 0 ; i < n; i++ )
{
    int min = i;
    for ( int j = i+1; j < n; j++ )
    {
        if ( A[j] < A[min] )  then  min = j;
    }
    if ( min != i )  then  swap( A[i], A[min] );
}
```

- Worst case: T(n) = $O(n^2)$ comparisons
- Best case: T(n) = $O(n^2)$ comparisons
- Average case: T(n) = $O(n^2)$ comparisons
- Worst-case space complexity: $O(1)$ auxiliary
- Overall algorithm: T(n) =$\theta(n^2)$ as $O(n^2)$ and $\Omega(n^2)$ are the same

Summary

- Efficient data structures and algorithms are crucial for successful computer applications.
- Measure runtime as a function of the given input size.
- Examine asymptotic behaviour and refer to complexity classes.