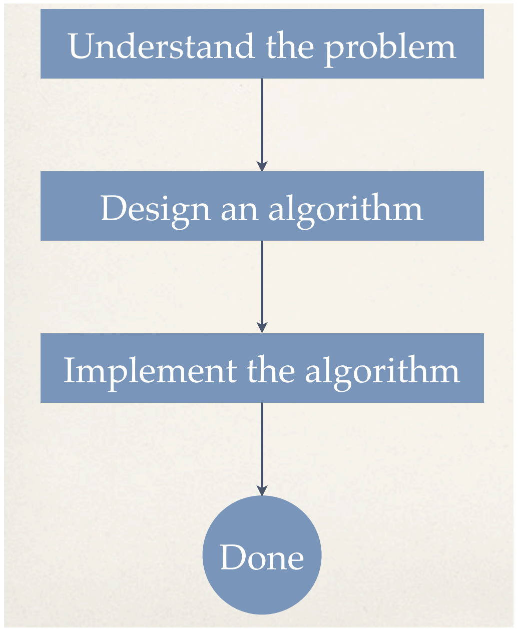
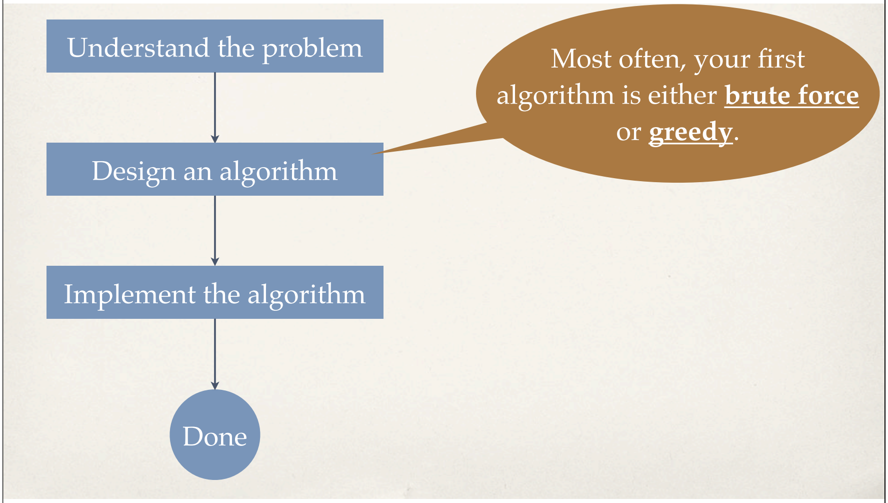
What we usually do?



slides by Haleh Ashki



## Common Types of Approaches for Designing Algorithms

- Brute Force
- Divide and Conquer
- Decrease and Conquer
- Transform and Conquer
- Greedy

## Brute Force

- a straightforward approach to solving a problem
- usually it is not the best way to solve a problem but it has the advantage that it is conceptually simple
- is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement
- Find all possible combinations of feasible solutions and pick the one which satisfies the given criteria.

## Examples

•Searching

If we have an array of names and we want to find the first occurrence of a particular name, say “smith”, then we compare “smith” with the first entry and if they are not equal we move to the second entry and compare it, etc. This is a brute force approach to searching.

• Sorting

If we have an array of real numbers then a brute force approach to sorting it in ascending order would be to look through the array and find the small entry and exchange it with the first entry in array; then search the second through last entries and find the smallest entry and exchange it with the second entry, etc.

## Use a brute force approach to sorting the array

a = **{**17, 31, 6, 4**}**

– On the first step we find that the smallest entry is in a(4) so we exchange a(1) and a(4) to get **{**4, 31, 6, 17**}**

– On the second step we look at entries two through four of the new array and see that the smallest is in position 4 so we exchange to get **{**4, 6, 31, 17**}**

– On the third step we see that the fourth entry is larger than the third so we exchange to get the final sorted array

**{**4, 6, 17, 31**}**.

## Divide and conquer

- The basic idea is to divide the problem into several smaller problems of the same type; each subproblem may be divided further.
- Probably the best known general algorithm design technique

## Example

As an example consider again the problem of *searching* an array. A brute force approach was the exhaustive approach of checking the first entry, then the second, etc. What would be Divide and Conquer approach?

Looking for a world in a dictionary or for a name in phone book. (what we usually do?)

If the array is ordered (say in ascending order or alphabetical order) then we could check the middle entry in the array and if it was not equal then we would know whether it was in the first half of the array or the last half because the array was ordered; so we have divided the problem into a smaller problem.

## Mathematical example

Finding the root of **f(x)** on **[a, b]**

**Bisection Method**

repeatedly bisects an interval and then selects a subinterval in which a root must lie for further processing

Explicitly, if $f(a)$ and $f(c)$ are opposite signs, then the method sets $c$ as the new value for $b$, and if $f(b)$ and $f(c)$ are opposite signs then the method sets $c$ as the new $a$. (If $f(c)=0$ then $c$ may be taken as the solution and the process stops.) In both cases, the new $f(a)$ and $f(b)$ have opposite signs, so the method is applicable to this smaller interval.

## Example

Find the location where 17 occurs in the sorted array

**a**0 = **{**1**,** 4**,** 7**,** 9**,** 17**,** 31**,** 33**}**.

– first we compare 17 with **a**0(4) and see that 17 **>** 9 so we know that 17 $\in$ **a**1 = **{**17**,** 31**,** 33**}**.

– next we compare 17 with **a**1(2) and see that 17 **<** 31 so 17 is in **a**2 = **{**17**}** and we have located the element.

This approach is called a ***Binary Search*** and is a common example of a divide and conquer approach.

## Decrease and conquer

- Steps to solve:

1.Reduce problem instance to smaller instance of the same problem

2.Solve smaller instance

3.Extend solution of smaller instance to obtain solution to original instance

- Probably the best known general algorithm design technique
- This strategy is based on exploiting the relationship between a solution to a problem of size n and a solution to a smaller problem.
- Can be implemented either top-down or bottom-up
- Also referred to as *inductive* or *incremental* approach

- As an example, consider calculating $\pi^8$.

Recall that the brute force approach was to compute $\pi * \pi * \pi * \pi * \pi * \pi * \pi * \pi$ which required 7 multiplications.

A decrease and conquer approach would be to note that $\pi^8 = \pi^4 \pi^4$. To compute $\pi^4$ we note that it is equal to $\pi^2 \pi^2$. Consequently we form $\pi^2$ (1 multiplication), then $\pi^4 = \pi^2 \pi^2$ (1 multiplication) and $\pi^8 = \pi^4 \pi^4$ (1 multiplication); thus we have computed the work in three multiplications rather than seven.

Of course we would have to modify the algorithm slightly if we wanted to compute $\pi^9$. In this case we would simply write $\pi^9 = \pi^8 * \pi$ and compute $\pi^8$ as above and then perform one additional multiplication to get $\pi^9$.

- A **bottom-up** approach (also known as inductive reasoning and in many cases used as a synonym of *synthesis*) is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. Like as lego!

- A **top-down** approach (also known as stepwise design or deductive reasoning and in many cases used as a synonym of *analysis* or *decomposition*) is essentially the breaking down of a system to gain insight into its compositional sub-systems.

function f(n)=a$^n$

using bottom up (iterative)approach we can multiply a by n-1 time of a.

**iterative:**

Input: an integer

Output: an integer

```
function F(a,n)
    x=1
    for i=1 to n:
        x = x * a
    end for
    return x
end function
```

An algorithm for calculating polynomials

$$p_3(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

$$p_3(x_0) = a_0 + a_1 \times x_0 + a_2 \times x_0 \times x_0 + a_3 \times x_0 \times x_0 \times x$$

That is $(1+2+3) = 6$ multiplications and 3 additions.

Horner's method (from algI)

$$\begin{aligned} p_3(x_0) &= a_0 + x_0\,(a_1 + x_0(a_2 + a_3 x_0)) \\ &= a_0 + x_0\,(a_1 + x_0(a_2 + b_3 x_0)) \\ &= a_0 + x_0(a_1 + x_0 b_2) \\ &= a_0 + x_0 b_1 \\ &= b_0 \end{aligned}$$

$$p_3(x_0) = a_0 + x_0 \times (a_1 + x_0 \times (a_2 + a_3 \times x_0))$$

That is 3 multiplications and 3 additions

So that is an iterative bottom up method

Efficiency:

Naïve method:

n(n+1)/2 (multiplications ) + n (additions)

Horner's: n (multiplications ) + n (additions) for **recursive**:

Using top down(recursive) approach we can write:

Input: an integer
Output: an integer

```
Function  F(a):
   if a>1:
      return a * F(a-1)
   return a
End Function
```

## Recursion

Factorial n!:

5!= 5 x 4 x3 x 2 x 1

5! = 5 x 4!

5! = 5 x 4 x 3!

5! = 5 x 4 x 3 x 2!

5! = 5 x 4 x 3 x 2 x 1!

5! = 5 x 4 x 3 x 2 x1

$$\operatorname{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \operatorname{fact}(n-1) & \text{if } n > 0 \end{cases}$$

**Pseudocode:**

```
function factorial is:

input: integer n such that n >= 0

output: [n × (n-1) × (n-2) × … × 1]

   1. if n is 0, return 1
   2. otherwise, return [ n × factorial(n-1) ]

end factorial
```

fact(5)

5 x fact(4)

5 x (4 x fact(3))

5 x (4 x (3 x fact(2)))

5 x (4 x (3 x (2 x fact(1))))

5 x (4 x (3 x (2 x 1)))

24

## Memory allocation and data storage

```
factorial(20) -- allocate 1 word of memory,
  call factorial(19) -- allocate 1 word of memory,
    call factorial(18) -- allocate 1 word of memory,
      .. . . .
      call factorial(2) -- allocate 1 word of memory,
        call factorial(1) -- allocate 1 word of memory,
          call factorial(0) -- allocate 1 word of memory,
at this point 21 words of memory and 21 activation records
have been allocated.
        return 1.      -- release  1 word of memory.
      return 1*1.      -- release  1 word of memory.
    return 2*1.      -- release  1 word of memory.
```

21

## 3 Types of Decrease and Conquer

- *Decrease by a constant* (usually by 1):
  - insertion sort
  - graph traversal algorithms (DFS and BFS)
  - topological sorting
  - algorithms for generating permutations, subsets
- *Decrease by a constant factor* (usually by half)
  - binary search and bisection method
  - exponentiation by squaring
  - multiplication à la russe
- *Variable-size decrease*
  - Euclid's algorithm
  - selection by partition
  - Nim-like games

22

## Transform and Conquer

In this approach we transform the problem into one which is more amenable to solution.

In Algorithms I you studied Gaussian elimination (GE) for solving a linear system. For GE you transform the linear system into an equivalent upper triangular system and we know that solving upper triangular systems is"easy".

This is a technique that is used throughout mathematics too. For example, when you calculate the integral $\int_D (x^2 + y^2)\ dxdy$ where $D$ is the unit circle it is much easier to transform the integral to polar coordinates using the transformation $x = \cos\theta$, $y = \sin\theta$ to obtain the equivalent integral $\int_0^{2\pi} \int_0^1 r^2 r\ dr d\theta$.

23

Suppose you wanted to see if any two elements of an array are equal.

**transform and conquer approach**

first sort the array (i.e., transform the problem). Now all we must check is to see if two adjacent entries of the sorted array are equal. If we use an efficient sorting routine then this approach will be faster than the brute force approach.

**Example**. Determine if any two entries of the array {61, 17, 32, 4, 17} are equal.

first sorting the array into

{4, 17, 17, 32, 61}

and then checks

– Is 4 = 17? Is 17=17? is 17=32? Is 32=61?

24

## Greedy

A **greedy algorithm** works in phases. At each phase:

You take the best you can get right now, without regard for future consequences.

the choice is irrevocable, i.e., it cannot be changed on subsequent steps of the algorithm.

You hope that by choosing a ***local*** optimum at each step, you will end up at a *global* optimum

This technique is not as broad as the others and is used for optimization problems.

An **example** of where a greedy algorithm might be useful is the "change problem" faced by cashiers all over the world where one wants to give the change using the criteria that we use as small a number of coins as possible.

## example

Use a greedy algorithm to determine the smallest number of coins needed to give the change of 43 cents assuming that the available coins are quarters, dimes, nickels and pennies.

– On the first step the available coins are quarter, dime, nickel and penny because all are less than 43 cents. We choose the largest one (we are greedy after all! ), a quarter and we now have 43-25=18 cents.

– On the second step the feasible coins are dime, nickel and penny and we choose the largest, a dime; we now have 18-10 = 8 cents.

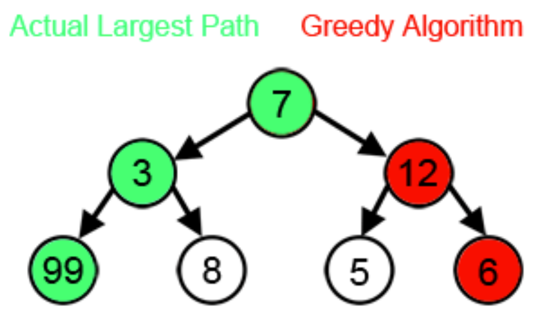
– On the third step the feasible coins are a nickel and a penny. We choose the largest which is a nickel and we have 8-5=3 cents.

– On the fourth step the only feasible coin is a penny.

– The optimal number of coins is 6 - a quarter, a dime, a nickel and three Pennies.

## When greedy fails

Goal: Reaching the largest sum



## How we can compare the algorithm

If we have two different algorithms that solve the same problem then how can we determine if one is **“better”** than the other?

We can compare the

- **storage** (e.g., the **size** of the arrays required , memory usage).
- **speed** (e.g., which one runs faster)
- **efficiency**

## Storage

The **size** of memory that is used for solving the problem.

The algorithm that use less space (memory) to do the same thing as another one is doing better performance.

Example: adding two arrays
We can creating a new array for result:
(1,2,3,4,5) + (1,2,3,4,5) = (2,4,6,8,10) : 3 arrays of size 5
Or replace the sum in a same spot of one of the values :
(1,2,3,4,5) + (1,2,3,4,5)
(2,2,3,4,5) + (1,2,3,4,5)
(2,4,3,4,5) + (1,2,3,4,5) No additional array is used.
(2,4,6,4,5) + (1,2,3,4,5)
(2,4,6,8,5) + (1,2,3,4,5)
(2,3,6,8,10) + (1,2,3,4,5)

## Speed(wall clock time)

If we run both algorithms for a particular problem and Algorithm A runs **faster** (i.e., it takes less wall clock time) than Algorithm B then we might conclude that Algorithm A is better. However, this might not be the case.

When we do the comparison we are performing the calculations for a specific value of the problem size (for example, searching an array of length 100). However, if we run the same algorithms for a different problem size (such as searching an array of length 100,000) then we might find that Algorithm B runs faster.

We also have to be concerned about how each algorithm is **implemented** and how issues like **initialization**, etc. are handled.

## Efficiency

Then what can we use to compare the efficiency of two algorithms?

Typically we would like to estimate the **work**, i.e., the number of operations performed as a function of a parameter that characterizes the **size of the problem**.

## What do we mean by the **size of the problem**?

Usually a problem size is a function of some parameter **n**. Some examples include:

– When we multiply a square matrix times a vector than the parameter is the size of the matrix, i.e., **n** where the matrix is **n x n**. We know that as **n** increases the number of arithmetic operations increases

– Another example would be sorting or searching a string of length **n**.

If we are solving a small instance of a problem then it probably doesn't matter whether we use the most efficient algorithm.

However, if we want to solve **large** problems (i.e., for large **N**) or we need to perform the calculation many times, then we have to be concerned about **storage** and the growth **rate of the work** in terms of **N**.

If we want to develop efficient algorithms then we must be able to state mathematically what we mean by "efficient"; we need to be able to say something more than "it runs quickly."

**The question is:**

how we can **quantify** the efficiency of algorithms so that we can compare them.

- The wall clock time that an algorithm takes to execute for a specific problem can depend on a lot of factors; for example, the actual implementation (coding), the language used, the computer used, etc.
- We want a definition of efficiency that is platform-independent, instance-independent and of predictive value as the input size is increased.

Analyzing algorithms involves thinking about how their resource requirements – the amount of time and space they use – will scale with increasing input size. In most cases, the value of one particular input quantity is a measure of how hard the calculation is going to be.

• Often this quantity is an integer, perhaps **N**, which might measure the length of an input vector, the dimension of a square matrix (i.e., **N x N**), the number of iterative steps to take, or some other quantity that affects the amount of work.

• It is sometimes possible to estimate the **work W**, the number of **operations performed**, as a function of an input parameter such as **N**.

Suppose we were able to determine an explicit formula involving N for the work required to use each of two methods (Algorithm A and Algorithm B)

to solve a problem and found these formulas to be

$\mathbf{W_A} = 3\mathbf{N} + 21 \qquad \mathbf{W_B} = \mathbf{N}^2 + 10\mathbf{N} + 5$

where $\mathbf{W_A}$ denotes the work for Algorithm A and $\mathbf{W_B}$ denotes the work for Algorithm B.

We want to investigate the implications of these two formulas remembering that we are concerned with how the **work grows as N increases**; if we are performing calculations with small values of N then it probably doesn't matter which algorithm we use.

The first thing to note in formulas like these is that as N grows the term which has the highest power of N dominates; for example, in $W_A$ it is **3**N and in $W_B$ it is N**²**.

| $N$ | $3N$ | $3N+21$ | $N^2$ | $10N$ | $N^2+10N+5$ |
|---|---|---|---|---|---|
| 10 | 30 | 51 | 100 | 100 | 205 |
| 100 | 300 | 321 | 10,000 | 1000 | 11,005 |
| 1000 | 3000 | 3021 | 1,000,000 | 10,000 | 1,010,005 |
| 100,000 | 300,000 | 300,021 | $10^{10}$ | $10^6$ | 10,001,000,005 |

We say that $\mathbf{W_A}$ is *linear* in **N** and $\mathbf{W_B}$ is *quadratic* in **N**.

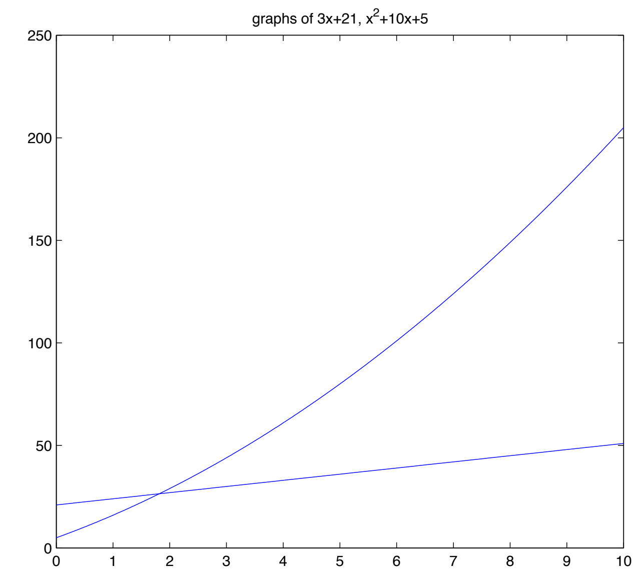
Next slide you se a plot of the two formulas for the work as a function of **N**. Note that this means that if **N** is doubled (say 1000 to 2000) then the work $\mathbf{W_A}$ increases by approximately two (from 3021 to 6021, i.e.,

from **3N + 21** to **3(2N) + 21**

However for $\mathbf{W_B}$ the work increases by approximately four; i.e., from

$\mathbf{N^2 + 10N + 5}$ to $\mathbf{(2N)^2 + 10(2N) + 5 = 4N^2 + 10N + 5}$

This gives us a measure of how complex the problem is in terms of **N**.



## Terminology

• When the leading term in the work is a constant times N we say the method
– has linear growth in N or equivalently
– is order N or equivalently
– is ***O(N)***

• When the leading term in the work is a constant times $N^2$ we say the method
– has quadratic growth in N or equivalently
– is order $N^2$ or equivalently
– is ***O($N^2$)***

## Polynomial Growth

- We have seen two examples of polynomial growth, linear which is $\mathcal{O}(N)$, and quadratic which is $\mathcal{O}(N^2)$.
- Clearly we could have work which has a leading term of $N^3$ and we would call this method $cubic$ and say it is $\mathcal{O}(N^3)$.
- So, in general, if a method has polynomial growth then we say it is $\mathcal{O}(N^p)$ for some $p > 0$ which is typically an integer but doesn't have to be.
- Remember that $\mathcal{O}(N^p)$ means that the leading term in the work is $c * N^p$ for some positive constant $c$.
- If the value of $N$ increases from $N$ to $2N$ then the amount of work increases by $2^p$ because we compare $cN^p$ and $c(2N)^p = c2^pN^p$.
- These methods are easy to compare because the larger the value of $p$ the more work required. We can compare these to the plots of the continuous monomials $x$, $x^2$, $x^3$, ... We know that as the power of $x$ increases the plot goes to infinity faster and faster.

Are there methods which have work which don't have polynomial growth?

- Suppose we determined that an algorithm has a formula for work which is

$$\log N + 5$$

- First of all we might wonder what this means because there is no base for the log function. Oftentimes in logarithmic growth formulas the base is omitted; this is because we can always change between bases by using the formula

$$\log_a x = \frac{\log_b x}{\log_b a}.$$

The denominator in this formula is a constant so if the method is $\mathcal{O}(\log_a N)$ then it is also $\mathcal{O}(\log_b N)$.

The function $\log N + 5$ is clearly not a polynomial but we might want to compare it to an algorithm which has polynomial growth. For example, does it require more work or less work than a method with linear or quadratic growth?

Other examples of logarithmic growth formulas are

$$N \log N \qquad N^2 \log N \qquad N(\log N)^2$$

We can also have exponential growth formulas such as

$$2^N \qquad 1.5^N \qquad 5^N$$

• We can also have a factorial growth N**!**. Note that

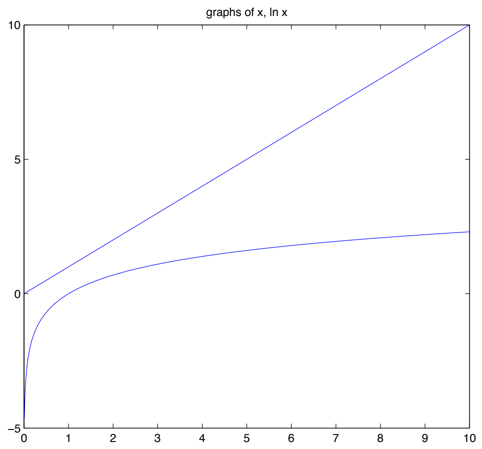
$$N! \approx N^N$$

due to Stirling's formula which for all practical purposes means it's impossible!

## How compare these formulas with polynomial growth?

One way to do this is to **plot** the corresponding continuous function (if appropriate).

For example, for lnN + 5 we could plot ln x + 5 and compare with polynomial growth. In the following plot we graph ln x + 5 and x.



- Another way to compare the growth is to use limits from calculus like:

$$\lim_{x\to\infty} \frac{\ln x}{x} \qquad \lim_{x\to\infty} \frac{2^x}{x^5}$$

L'Hôpital's rule

$$\lim_{x\to c} \frac{f(x)}{g(x)} = \lim_{x\to c} \frac{f'(x)}{g'(x)}$$

## Example

$$\lim_{x\to\infty} \frac{\ln x}{x} = \lim_{x\to\infty} \frac{\frac{1}{x}}{1} = \lim_{x\to\infty} \frac{1}{x} = 0$$

which says that x approaches infinity faster than ln x does. This means that a method which is linear in growth requires **more work** than a method which has logarithmic growth. This is exactly what we concluded from our graph above.

$$\lim\inf \frac{2^x}{x^5} = \lim\inf \frac{2^x \ln 2}{5x^4} = \lim\inf \frac{2^x \ (ln2)^2}{5x^4} = \ldots = \lim\inf \frac{2^x \ (ln2)^5}{120}$$

which says that **2**$^x$ grows faster than x$^{\mathbf{5}}$. Note that this is also true for x$^p$ for any p >= **0**. We say that **2**$^x$ has exponential growth. Exponential functions growth faster than any polynomial.

To compare two exponential growth formulas, such as $a^N$ and $b^N$ we simply look at the base; if $a > b$ then $a^N$ grows faster.

It is important to realize that what we are interested in is the rate of growth. If we have two algorithms which have work $3N^2 + 4$ and $4N^2 + 4$ it is true that for any **N** the work for the first is less than for the second but the rate at which they grow is the same. For example, for $N = 10^4$ they both have $O(10^8)$ operations.

49

| | $N$ | $N \log_2 N$ | $N^2$ | $N^3$ | $1.5^N$ | $2^N$ | $N!$ |
|---|---|---|---|---|---|---|---|
| $N = 10$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 4sec |
| $N = 30$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 18 min | $10^{25}$ yrs |
| $N = 50$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | $< 1$ sec | 11min | 36 yrs | $\infty$ |
| $N = 100$ | $< 1$ sec | $< 1$ sec | $< 1$ sec | 1 sec | 12892 yrs | $10^{17}$ yrs | $\infty$ |
| $N = 10^3$ | $< 1$ sec | $< 1$ sec | 1 sec | 18 min | $\infty$ | $\infty$ | $\infty$ |
| $N = 10^4$ | $< 1$ sec | $< 1$ sec | 2 min | 12 days | $\infty$ | $\infty$ | $\infty$ |
| $N = 10^5$ | $< 1$ sec | 2 sec | 3 hrs | 32 yrs | $\infty$ | $\infty$ | $\infty$ |

Estimated running times of different algorithms on inputs of increasing size for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ the time is listed as $\infty$. Reference: Algorithm Design by Kleinberg & Tardos

50

If we break our algorithm into two parts and Part I is linear in N and Part II is linear in N then the algorithm is **linear**.

If we break our algorithm into two parts and Part I is linear in N and Part II is quadratic in N then the algorithm is **quadratic**

51

**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.

52

## 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

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

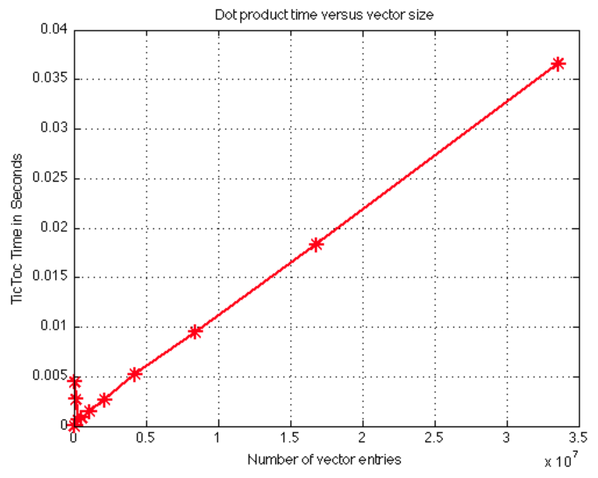
where $u_i$ denotes the $i$th entry of the vector $\vec{u}$.

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.

53



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.

54

## 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"!

55

## Floyd's algorithm for shortest path problem

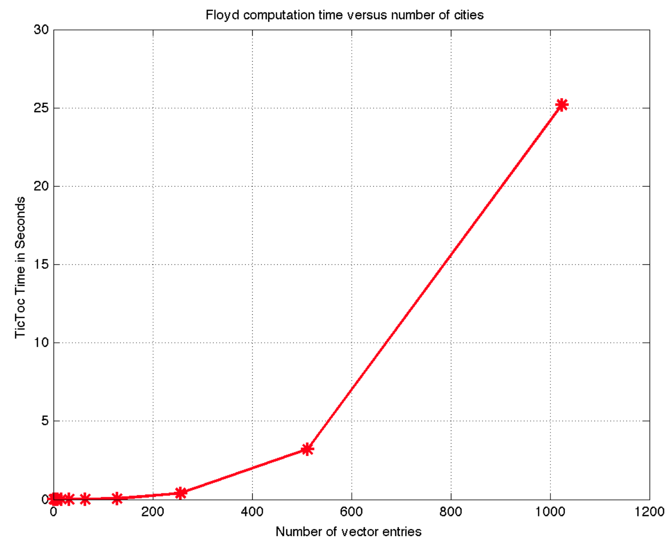
Instead of being impossible, Floyd's algorithm shows a simple way to compute

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**?

56



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 $n$th 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.