Introduction to Algorithms

- What is an algorithm?

  Here are some definitions:

  > *An algorithm is a description of a procedure which terminates with a result.*
  >
  > *An algorithm is a step-by-step problem-solving procedure, especially an established, recursive computational procedure for solving a problem in a finite number of steps.*
  >
  > *An algorithm is a sequence of unambiguous instructions for solving a problem by obtaining a required output for any legitimate input in a finite amount of time.*

  – The term *unambiguous* can not be stressed enough - we must be precise! For example, consider the following description of multiplying two $n \times n$ matrices, A, B. Why is it not clear? What should we add?

multiply each row of matrix A times each column of matrix B

This doesn't tell someone what the input or output is, how to form the multiplication, etc. It would be much clearer to write it as the following

Input: $n \times n$ matrices A, B
Output: an $n \times n$ matrix C which contains the product of A and B

```
for i = 1, n
    for j=1, n
        c(i,j) = $\sum_{k=1}^{n}$a(i,k)*b(k,j)
```

– Note that we add the caveat that it must work for any *legitimate* input. For example, if we are writing a routine for calculating the square root of a real number, we don't expect it to work for a negative number. Usually we code a test to make sure that the input is legitimate so we have a "nice" error message.

- Why do we need to study algorithms?

  Algorithms are the basis of computer programs and the computations generated by them are now used throughout society. For example, airplane wings are now designed using computers, decisions made concerning global issues such as climate change, groundwater contamination, etc. all rely on computer simulations. In fact, computations have joined theory and experimental as the pillars of scientific discovery.

- What are the goals we are setting for Algorithms I & II?
  - to learn a standard set of algorithms from different areas of computational science;
  - to see how these algorithms can be used to solve standard problems in scientific computing;
  - to be able to analyze algorithms as to efficiency, accuracy, and convergence;
  - be able to compare algorithms as to efficiency and accuracy;
  - to begin to see how to design new algorithms.
  - to understand the difference between continuous and discrete problems.

- What is the main difference between the courses Algorithms I & II?

  In Algorithms I we are concerned with numerical problems which typically involve mathematical objects of a *continuous* nature such as approximating an integral, solving a system of linear equations, finding the roots of a function, solving a differential equation, etc.

  In Algorithms II we are mainly interested in problems of a *discrete* nature such as searching for a text string, sorting a list of objects, finding the optimal path between cities, finding the point from a list which is closest to a given point, simulating a random process, etc.

Discrete vs. Continuous

The main distinction between Algorithms I and II is that the first deals with continuous problems and the second mainly deals with discrete problems. What do we mean by this?

- Real numbers have the property of varying smoothly so when we integrate a function $f(x)$ from $x = a$ to $x = b$ we expect f to take on all values between a and b.
- In contrast, the objects studied in discrete mathematics (such as integers, graphs, logical statements, etc.) do not vary smoothly in the same way that real numbers do. In fact they have distinct, separated values.
- Classic examples of discrete problems are sorting, searching, optimization using a set of discrete objects, operations dealing with images, etc. Graphs are often used to analyze discrete problems.
- However, with today's technology the distinction between continuous and

discrete sometimes becomes blurred. For example display monitors are an array of dots and are therefore discrete. However, the dots are so tiny that the depiction of a continuous function such as $\sin x$ looks like a continuous function on the screen. Another example is the number of colors available to view your photos on a screen. The number is so large that for all practical purposes it looks like we are using a continuous color spectrum.

- We will see that the algorithms to solve discrete problems are different from those for solving continuous problems but some of them have aspects in common with algorithms for continuous problems.
- As is the case for continuous problems, there will be several algorithms to solve a particular problem. Usually no one algorithms works for all instances of the problem.

Describing algorithms using pseudocode

- We need a clear and concise way to describe an algorithm which is not language dependent.
- Here we will use something called pseudocode which is a combination of common language and terminology typically used in computer languages such as loops and conditionals.
- It does NOT contain correct syntax for any language because it is language independent.
- However, we will often use Matlab syntax in this class for clarity.
- For a loop we will use the terminology `do` and `for` interchangeably.
- The following is an example of an algorithm written in pseudocode. Note that it does not use correct Matlab syntax and it uses common terms like "swap". The goal of writing the algorithm in pseudocode is to allow you to understand precisely what the steps of the algorithm are.

Example of pseudocode

```
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
```

Some Important Types of Problems

1. Sorting

Examples include

- a list of names which we have to sort alphabetically;
- a vector consisting of numbers which must be rearranged to appear in ascending order;
- a deck of n cards to be shuffled;
- sorting a list of students by their GPA;
- sorting a list of TVs by cost.

Sorting algorithms often require extra memory.

There is no single sorting algorithm that is best in all cases.

2. Searching

Examples include

- searching a protein string to identify the amino acid sequence that defines the protein

$$\mathrm{ATCGTATTGCACATTCTACGGGTAAATGCA}$$

- searching for a given value x in a numerical array $a(1:n)$;
- given a set of points X, find the point z in X which is closest to some given point,
- determining if any two elements in an array are equal

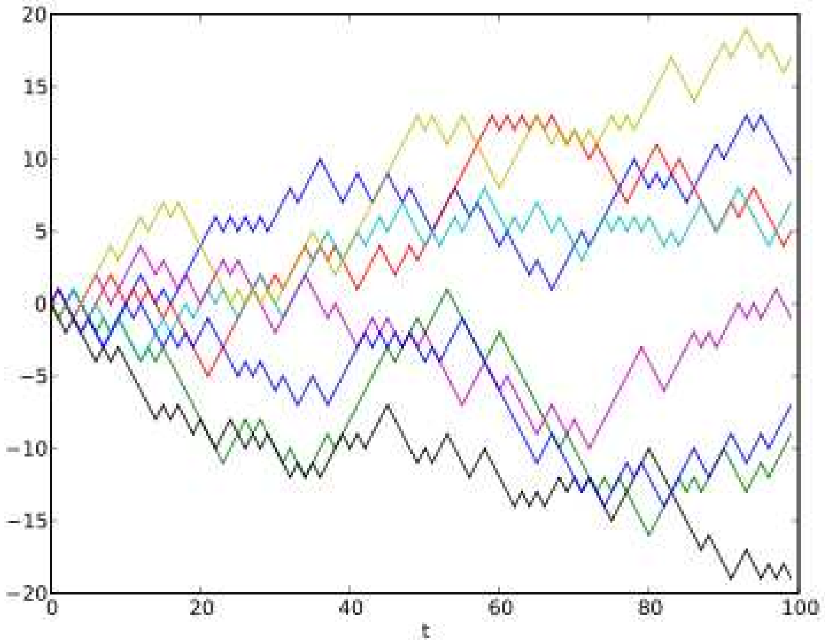
If we are searching a list and it is already sorted, then our algorithm should take advantage of this.

There is no single searching algorithm that is best in all cases.

3. Randomness

Examples include

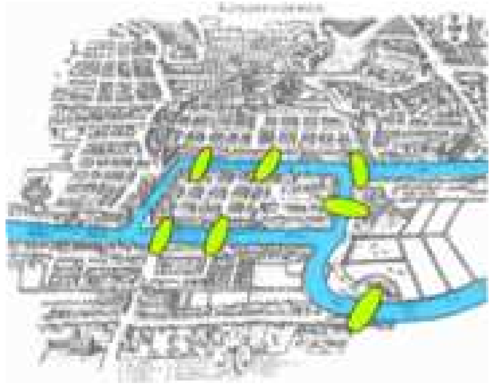
- Generating pseudorandom numbers
- Random walks - stock market, path of a foraging animal

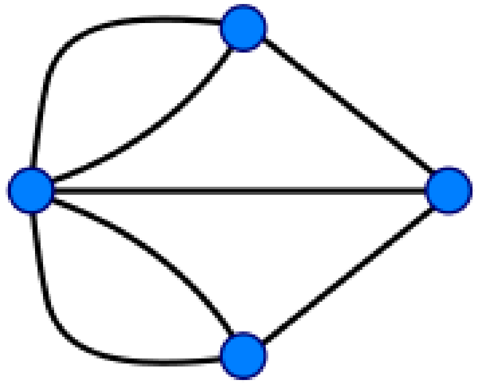


- Brownian motion

4. Graph Problems

- Königsberg Bridge Problem (1735)





- Minimum spanning tree

  An example would be a cable TV company laying cable to a new neighborhood. If it is constrained to bury the cable only along certain paths, then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house. There might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost.

5. Optimization Problems

Examples include

- Traveling Salesman Problem

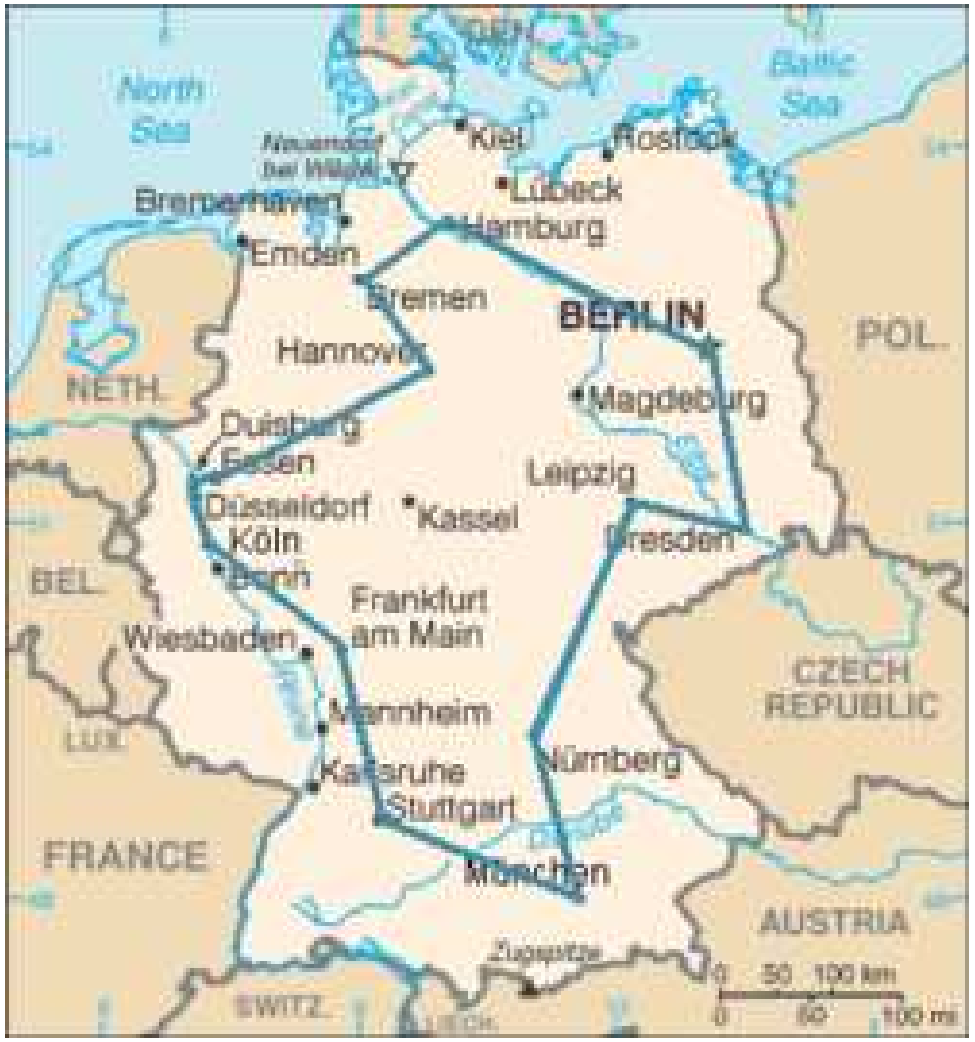


Fig. 0.1. *Shortest route visiting 15 cities among approximately 44 billion choices*

- Knapsack Problem

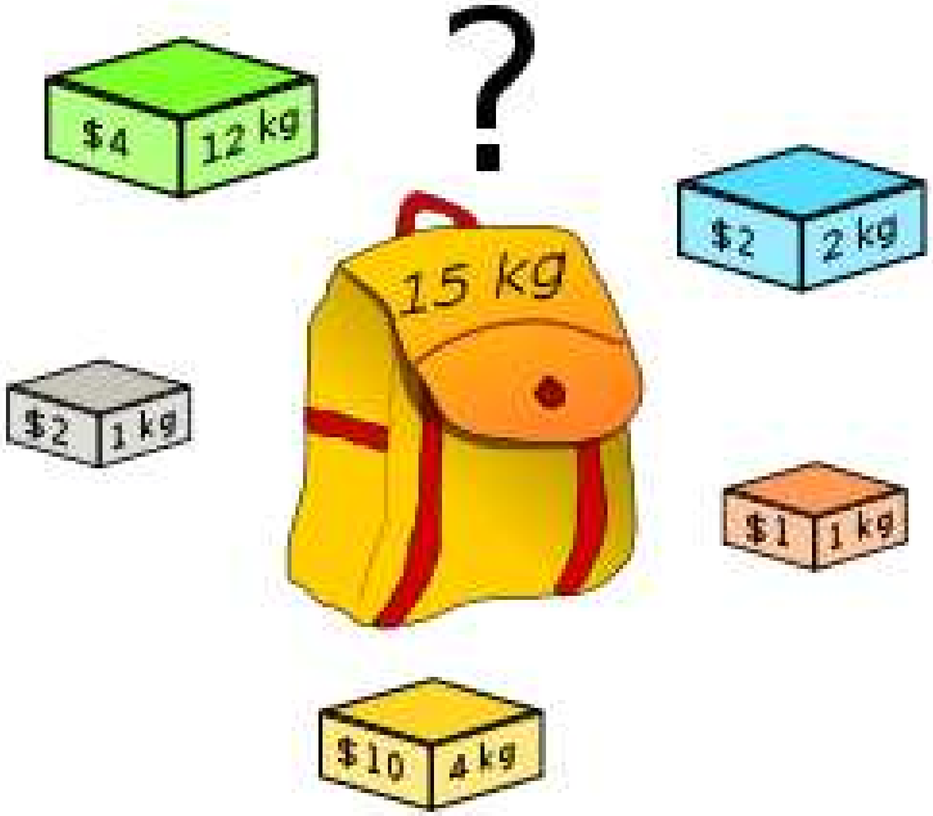


FIG. 0.2. *Maximize cost and minimize weight*

6. Data Mining & Clustering Problems

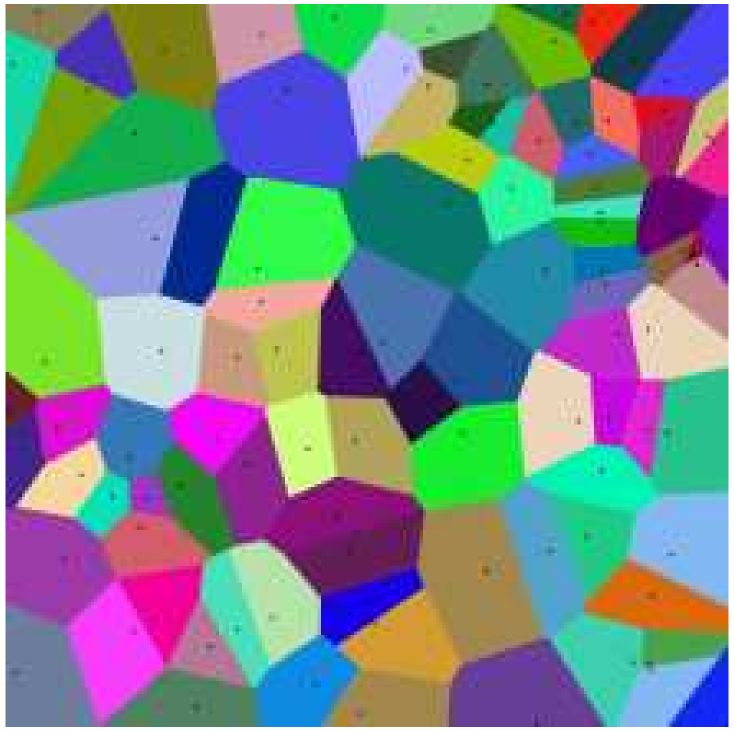
Examples include

- Law enforcement - fraud detection, criminal profiling
- Census data
- Medical statistics
- Stock market

7. Computational Geometry Problems

Examples include

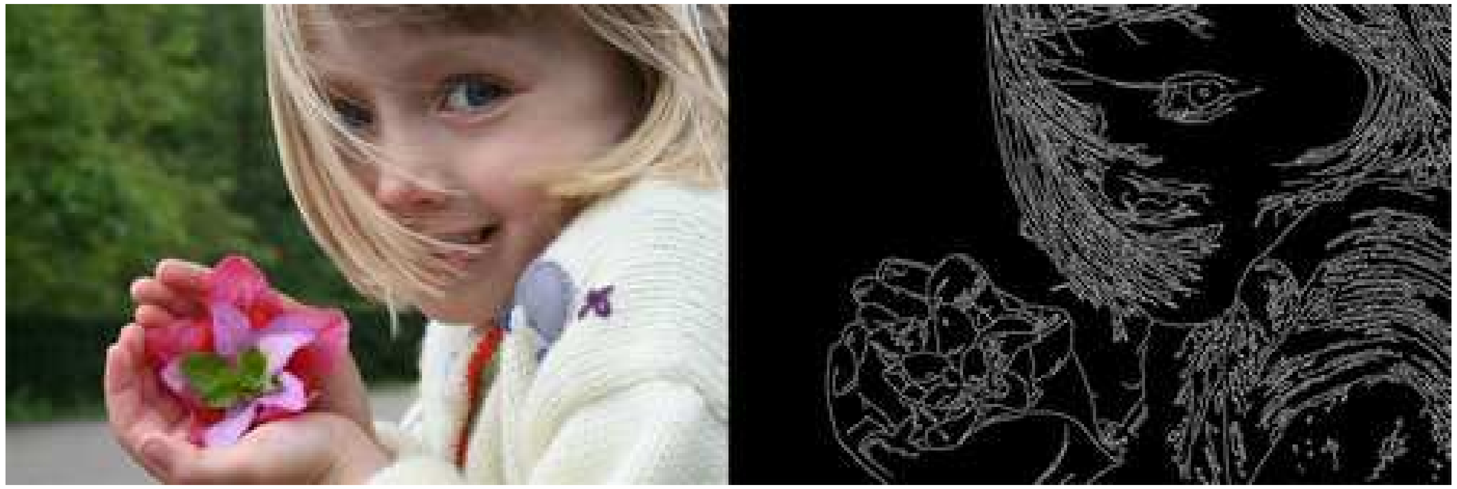
- Grid generation
- Convex Hull Problem - given a set of points, find the smallest convex polyhedron/polygon containing all the points.
- Voronoi Diagrams



8. Image Processing Problems

Examples include

- Feature extraction - facial recognition, fingerprint matching, etc.
- Edge detection - medical imaging
- Image enhancement
- Printing



9. Numerical Problems

These problem involve mathematical objects of a continuous nature and were studied in Algorithms I. Examples include

- solving systems of linear equations;
- computing a definite integral;
- solving a differential equation;
- finding the roots of a function (i.e., where it is zero);
- obtaining a simple function (such as a polynomial) to approximate a more complicated function

Common Types of Approaches for Designing Algorithms

1. 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
- If we have an array of names and we want to find the first occurence 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.
- 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.

- Find all possible combinations of feasible solutions and pick the one which satisfies the given criteria.

  Example 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 $\{4, 6, 17, 31\}$.

- If we wanted to compute a^n then a brute force algorithm is the following

  Brute Force algorithm for calculating a^n

```
Given $a$, $n$

value = 1.

for k = 1:n

    value = value * a
```

2. Divide and Conquer

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

  However, 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. You probably encountered this approach in a continuous setting if you used the Bisection Method to find the root of $f(x)$ on $[a, b]$.

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.

3. Decrease and Conquer

- This strategy is based on exploiting the relationship between a solution to a problem of size n and a solution to a smaller problem.
- As an example, consider calculating π^8.

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

  A decrease and conquer approach would be to note that $\pi^8 = \pi^4 \pi^4$. To compute π^4 we note that it is equal to $\pi^2 \pi^2$. Consequently we form π^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 π^9. In this case we would simply write $\pi^9 = \pi^8 * \pi$ and compute π^8 as above and then perform one additional multiplication to get π^9.

4. Transform and Conquer

- In this approach we transform the problem into one which is more amenable to solution.
- 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 $\int_0^{2\pi} \int_0^1 r^2 r\ drd\theta$.
- 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".
- Suppose you wanted to see if any two elements of an array are equal. The brute force approach is to check the first entry with the second through last entries. Then we check the second entry with the third through last entries, etc.

  An alternate approach which uses the transform and conquer approach is to 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.

The Brute Force approach checks the following:

$-$ Is 61 $=$17? Is 61 $=$ 32? Is 61 $=$ 4? Is 61 $=$ 17?
$-$ Is 17 $=$ 32? Is 17 $=$ 4? Is 17 $=$ 17?
$-$ Is 32 $=$ 4? Is 32 $=$17?
$-$ Is 4 $=$ 17?

The Transform and Conquer algorithm first sorts the array into $\{4, 17, 17, 32, 61\}$ and then checks

$-$ Is 4 $=$ 17? Is 17$=$17? is 17$=$32? Is 32$=$61?

5. Greedy Algorithms

- The strategy for these algorithms is to construct a solution through a sequence of steps where at each step the choice is made based upon the criteria that

  (i) it is the best local choice among all feasible choices available at that step and
  (ii) the choice is irrevocable, i.e., it cannot be changed on subsequent steps of the algorithm.

- 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.

What is a brute force approach to this problem?

How can we compare algorithms?

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).
- 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.

- 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 $n \times n$. We know that as n increases the number of arithmetic operations increases and you may have seen last semester that the leading term in the number of operations is n^2.
  - Another example would be sorting or searching a string of length n.

- In the next lecture we want to see how we can quantify the efficiency of algorithms so that we can compare them.

Analysis of Algorithm Efficiency

- 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 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., $N \times N$), 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.
- We look at situations where we can estimate the work required based upon an input parameter N and see how we can use this to compare algorithms. We need to return to calculus to help us understand how different formulas for work scale with N. When we encounter specific algorithms we will see how we can, in some cases, obtain an explicit formula for the work.
- 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

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

where W_A denotes the work for Algorithm A and 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 $3N$ and in W_B it is N^2. To see this, look at the following tables.

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

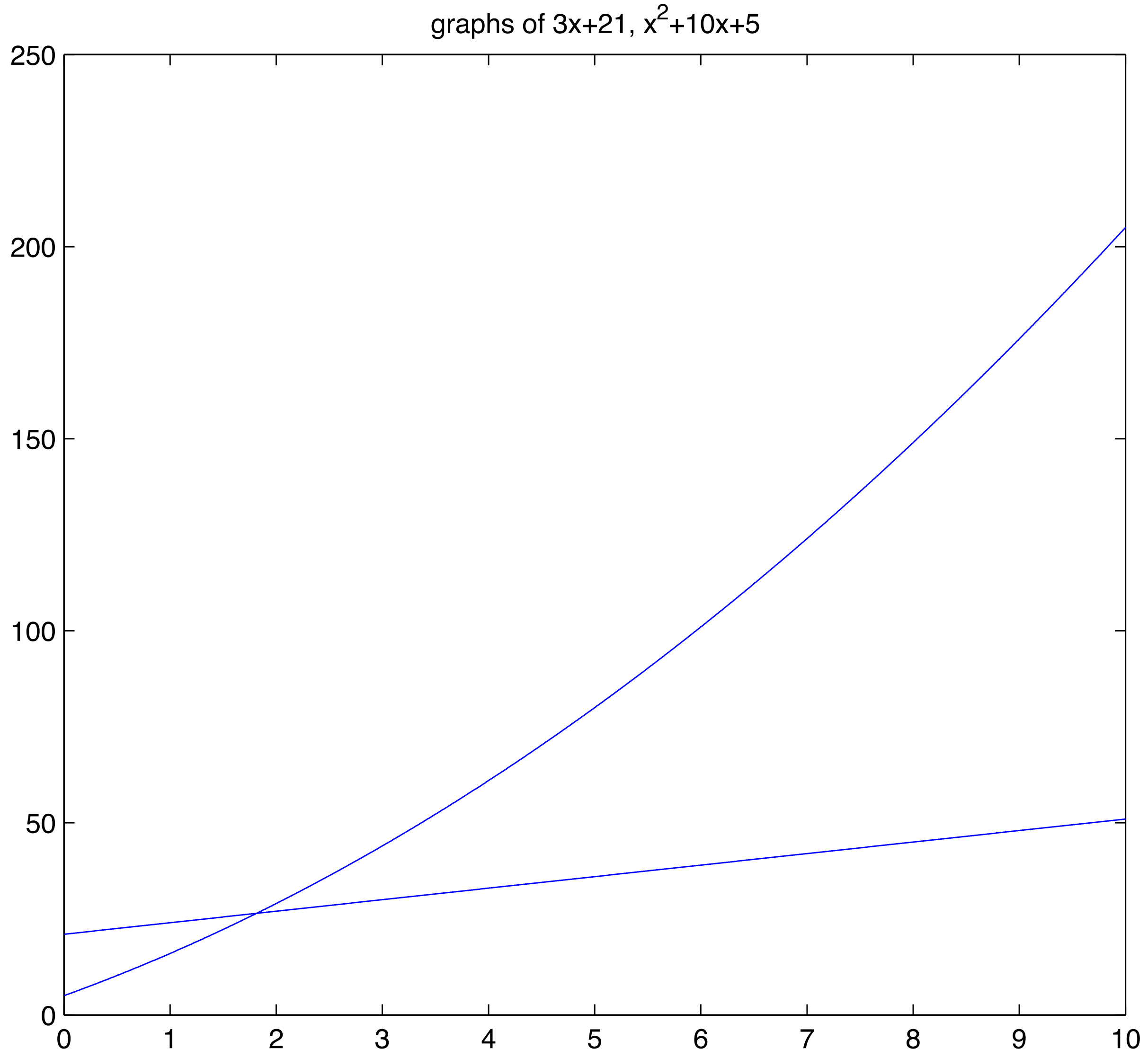
- We say that W_A is *linear* in N and W_B is *quadratic* in N. Here is a plot of the two formulas for the work as a function of N. Note that this means that if N is doubled (say 1000 to 2000) then the work W_A increases by approximately two (from 3021 to 6021, i.e., from

  $$3N+21 \qquad \text{to} \qquad to 3(2N)+21$$

  However for W_B the work increases by approximately four; i.e., from

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

  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 $\mathcal{O}(N)$

- 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 $\mathcal{O}(N^2)$

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, $\ldots$ 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

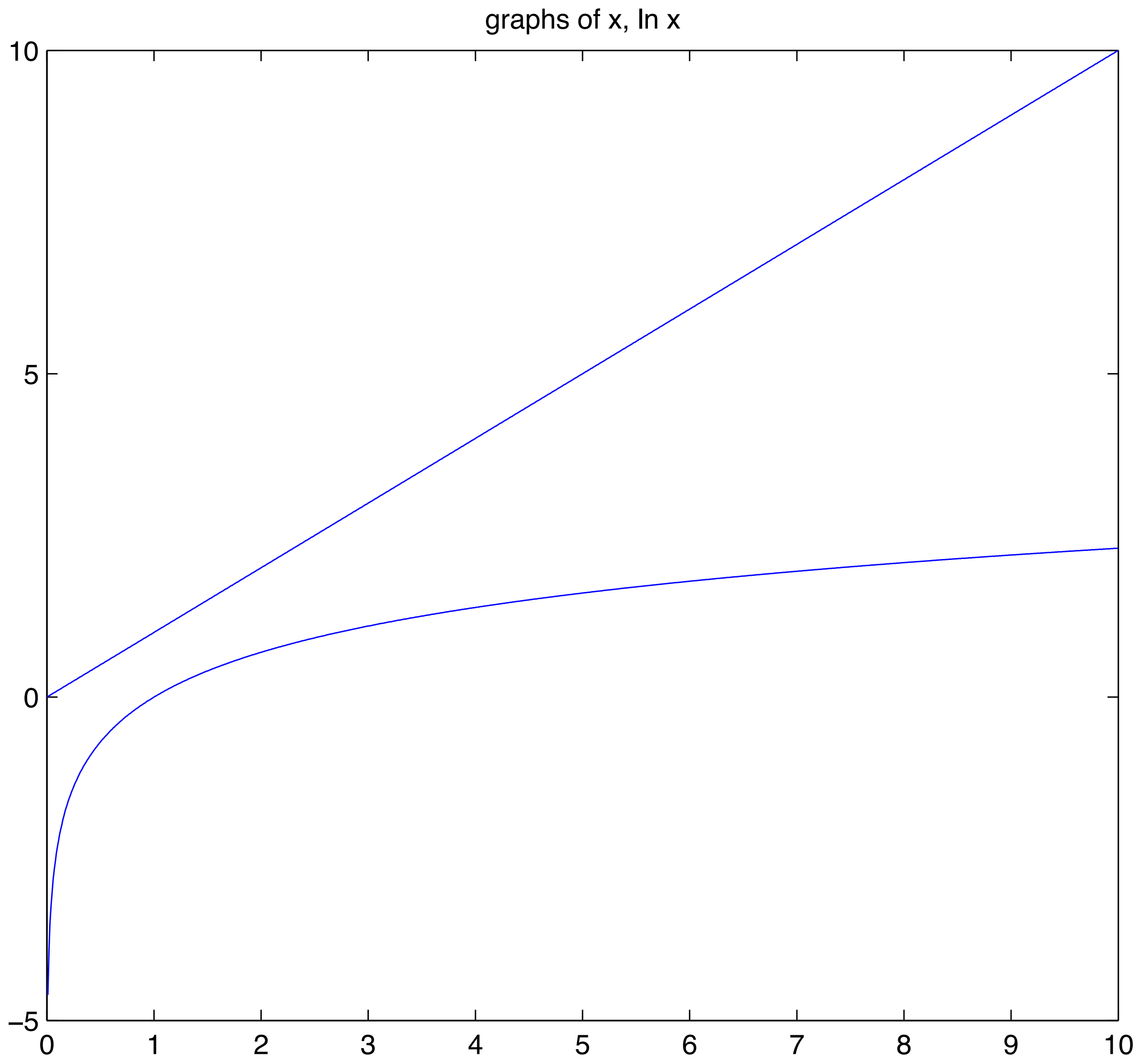
$$2^N \qquad 1.5^N \qquad 5^N$$

- 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!
- We want to compare these formulas with polynomial growth. One way to do this is to plot the corresponding continuous function (if appropriate). For example, for $\ln N + 5$ we could plot $\ln x + 5$ and compare with polynomial growth. In the following plot we graph $\ln x + 5$ and x. What can you conclude from this plot?

- Another way to compare the growth is to use limits from calculus and in particular l'Hôptial's rule. Remember in calculus that you were asked to evaluate limits like

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

In both of these limits you get an indeterminant form ∞/∞ and so you can



apply l'Hôptial's rule to get

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

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. For the other limit we have

$$\lim_{x\to\infty} \frac{2^x}{x^5} = \lim_{x\to\infty} \frac{2^x \ln 2}{4x^4} = \cdots = \lim_{x\to\infty} \frac{2^x (\ln 2)^5}{24} = \infty$$

which says that 2^x grows faster than x^5. Note that this is also true for x^p for any $p \geq 0$. 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.
- In the homework you will be asked to make a table of values for polynomial growth, logarithmic growth, exponential growth, etc.
- 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

$\mathcal{O}(10^8)$ operations.

	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	∞
$N = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12892 yrs	10^{17} yrs	∞
$N = 10^3$	< 1 sec	< 1 sec	1 sec	18 min	∞	∞	∞
$N = 10^4$	< 1 sec	< 1 sec	2 min	12 days	∞	∞	∞
$N = 10^5$	< 1 sec	2 sec	3 hrs	32 yrs	∞	∞	∞

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 ∞. Reference: Algorithm Design by Kleinberg & Tardos

- 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.

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.

Scalar or dot product of two vectors.

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

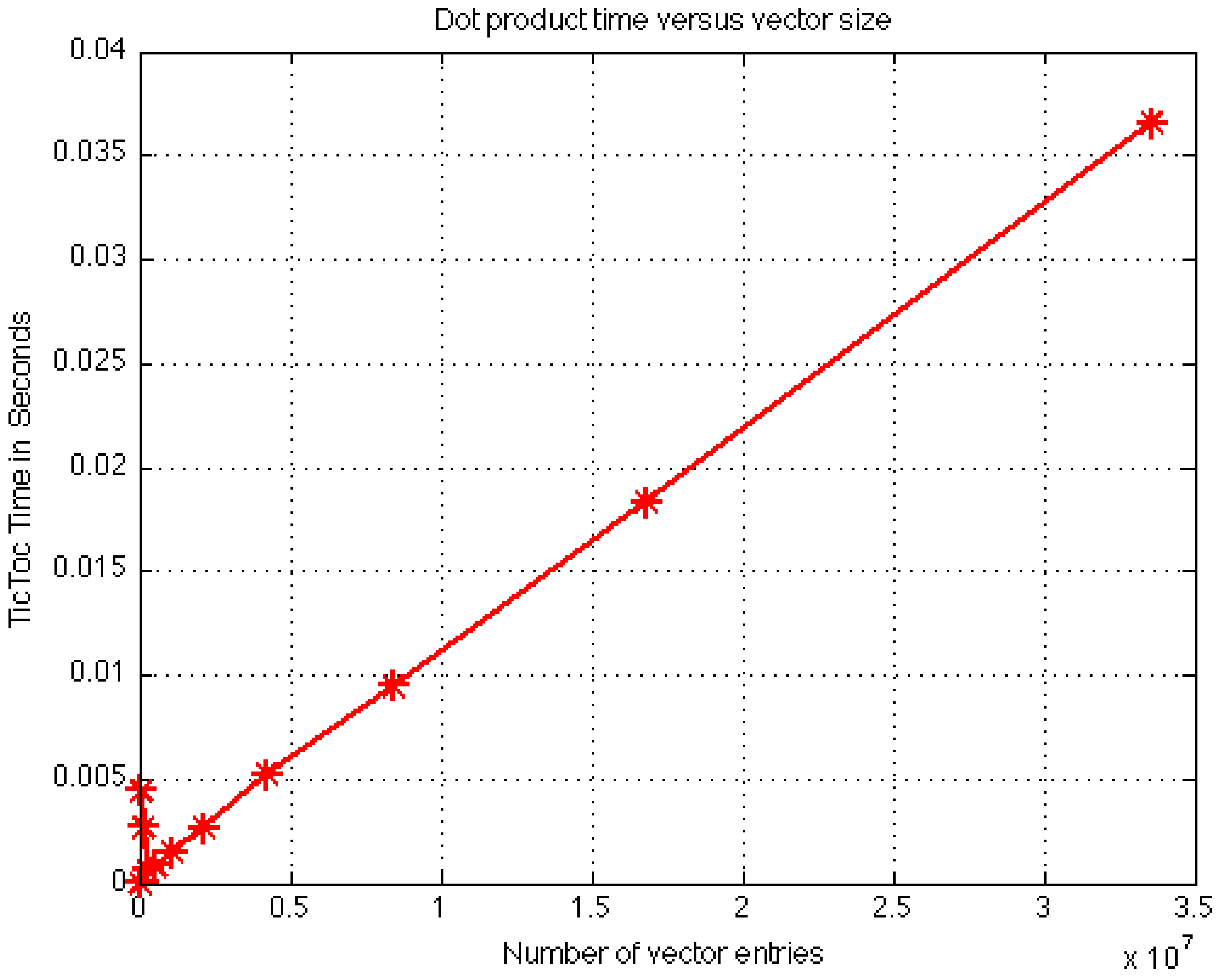
$$\vec{u}^T\vec{v} = \vec{u} \cdot \vec{v} = \sum_{i=1}^{n} u_i v_i$$

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.

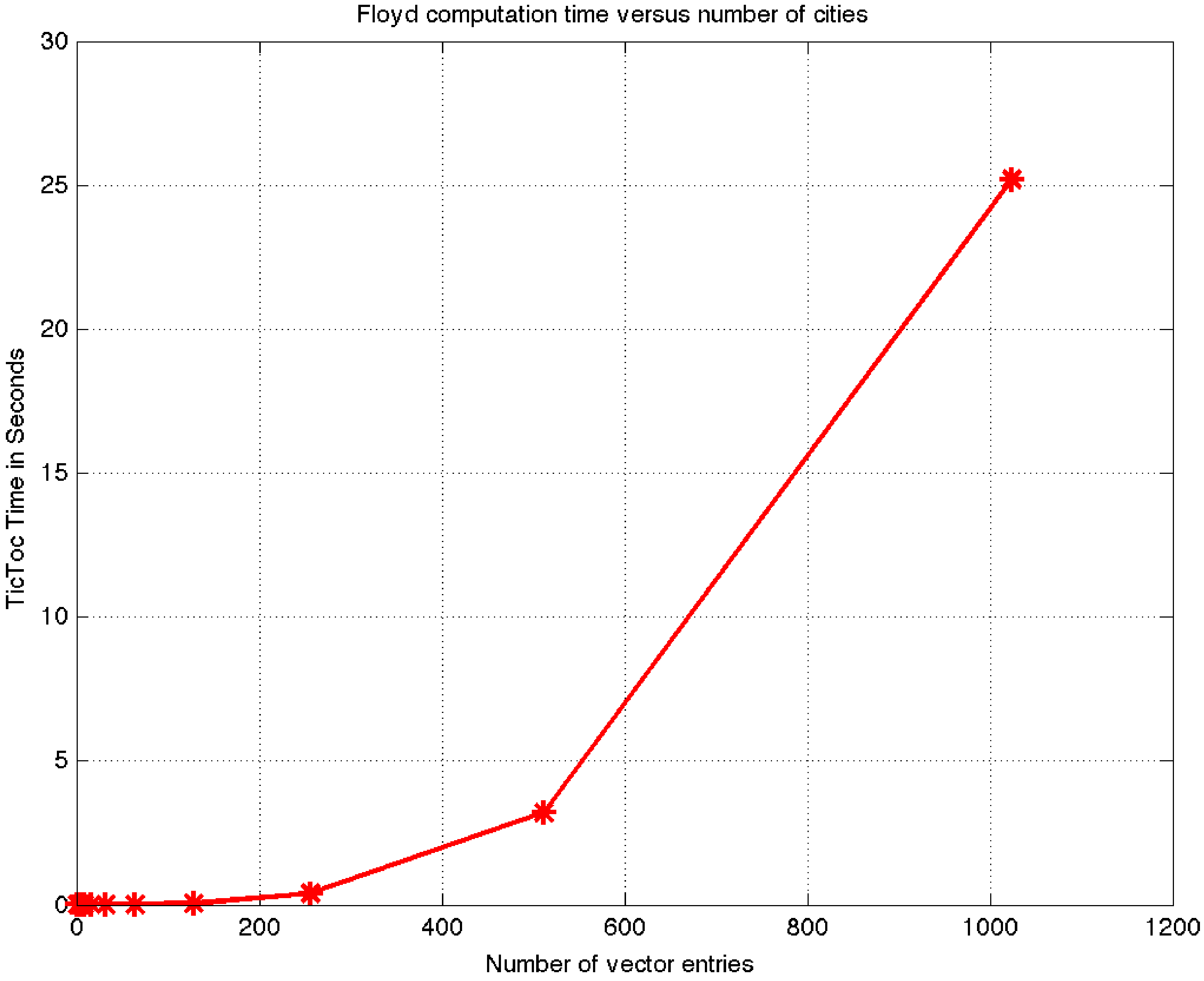
- We assume that we start with a table that gives the driving time $dt(i,j)$ for a direct trip from city i to each city j.
- If there is a direct route from city i to city j then it is easy. However many cities may not have a direct link. Usually there are many routes from one city to another and we 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

- In middle school you were probably asked to find the largest integer that divides two number evenly; 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 $\leq \min\{m, n\}$.
  - 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   Use this brute force algorithm to find gcd(16,24).

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

$t = 16$ 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.

Example 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.

In your first lab you will implement both of these algorithms and apply them to a problem.

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.

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.

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
```

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 $\mathcal{O}(n^2)$.
- 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. In the next class we will see an algorithm for sorting which is $\mathcal{O}(n \log n)$ and thus more efficient.

Bubble Sort

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.

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
$61 > 19$ 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.

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
```

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 'miami'.
- The brute force approach is to be given a list say a and a search key say $\mathcal{K}$.
  - Check if $a(1) = \mathcal{K}$; if so terminate, otherwise continue.
  - Check if $a(2) = \mathcal{K}$; if so terminate, otherwise continue.
  - Continue until one finds i such that $a(i) = \mathcal{K}$ or the list is exhausted.

Sequential Search Algorithm

Input: an array $a(1:n)$ and a search key $\mathcal{K}$

Output: the index of the first element of a that matches $\mathcal{K}$ or 0 if no match

```
i=0
while i $<$ n and a(i) $\neq \mathcal{K}$ do
    i $\leftarrow$ 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 whereas the best case scenario is $\mathcal{O}(1)$ 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.

Recall the problem of finding the shortest route to travel to 15 cities in Germany as depicted in the figure.

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.

Shortest route between 15 cities.

North Sea
Baltic Sea
Neuendorf bei Wilster
Kiel
Rostock
Lübeck
Bremerhaven
Hamburg
Emden
Bremen
BERLIN
POL.
Hannover
NETH.
Magdeburg
Duisburg
Essen
Leipzig
Düsseldorf
Kassel
Köln
Dresden
BEL.
Bonn
Frankfurt am Main
Wiesbaden
CZECH REPUBLIC
Mannheim
LUX.
Nürnberg
Karlsruhe
Stuttgart
Donau
FRANCE
München
AUSTRIA
Zugspitze
0 50 100 km
0 50 100 mi
SWITZ.
LIECH.

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

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

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

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

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

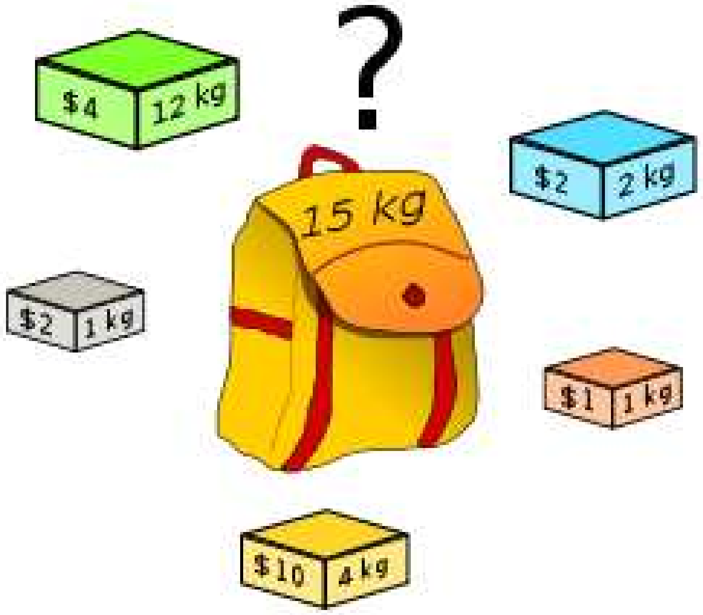
$A \rightarrow D \rightarrow B \rightarrow C \rightarrow A$=110+60+40+70=280

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

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

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 $\Longrightarrow$ 12 kg and \$4 value

1 green + 1 blue $\Longrightarrow$ 14 kg and \$6 value

1 green + 1 blue + 1 brown $\Longrightarrow$ 15 kg and \$7 value

1 green + 1 red $\Longrightarrow$ 13 kg and \$5 value

etc.

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

Describe a brute force algorithm for the following problems:

- The change problem - given an amount of change, e.g., 61 cents, determine the smallest number of coins one can use where the possibilities are quarter, dime, nickel and penny
- Closest pair problem - given a set X of n points we want to find the two points in X that are closest using the standard Euclidean distance formula

  $$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

  where the points are denoted (x_k, y_k).

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

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

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

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.

Lets consider the last step where we want to merge the two sorted arrays of length 4 above. Let $u =$ (12, 19, 45, 61), $v =$ (4, 22, 33, 71). 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;
- For our problem we have the following steps
  - $i = j = k = 1$, $u(1) > v(1)$ implies $w(1) = v(1) = 4$,$j = 2, k = 2$
  - $i = 1$, $j = k = 2$, $u(1) < v(2)$ implies $w(2) = u(1) = 12$, $i = 2$, $k = 3$
  - $i = j = 2$, $k = 3$ $u(2) < v(2)$ implies $w(3) = u(2) = 19$, $i = 3$, $k = 4$
  - $i = 3$, $j = 2$, $k = 4$ $u(3) > v(2)$ implies $w(4) = v(2) = 22$, $j = 3$, $k = 5$
  - etc.
- When $i = n$ or $j = n$ then we copy the rest of the remaining array into w.

Mergesort

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)
```

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

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

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.

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.

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

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 $\mathcal{K} <$ a(1) or $\mathcal{K} >$ a(n) return 0
left=1; right =n
while left $\leq$ right do
    m=(left+right)/2
    if a(m)=$\mathcal{K}$ return m
    if a(m) $> \mathcal{K}$
      set right=m
    else
      set left = m
end while
```

What about the efficiency of Binary Search? Is it $\mathcal{O}(n)$? Recall that if it is $\mathcal{O}(n)$ 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 $\mathcal{O}(\log n)$.

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 $\mathcal{O}(n^2)$.
- Can we design an algorithm which has fewer operations than $\mathcal{O}(n^2)$? 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 $a = a_1 a_0$, $b = b_1 b_0$ 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,

$$a = a_5 a_4 a_3 a_2 a_1 a_0 \qquad b = b_5 b_4 b_3 b_2 b_1 b_0$$

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

$$\alpha_1 = a_5 a_4 a_3 \quad \alpha_0 = a_2 a_1 a_0 \qquad \beta_1 = b_5 b_4 b_3 \quad \beta_2 = b_2 b_1 b_0$$

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 $(41*34)*10^4+(68*90-41*34-27*56)*10^2+\big(27*56\big)$

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 $(41*34)*10^4+(68*90-41*34-27*56)*10^2+\big(27*56\big)=1394*10^4+(6120-1394-1512)*10^2+1512=13,940,000+3214(100)+1512=13,941,512+321,400=14,262,912$

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.

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

$$a^n = [a^{n/2}]^2$$

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

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

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

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.

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

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 $\geq$ 0 and a(j)$>$ v
      a(j+1) $\leftarrow$a(j)
      j=j-1
    end while
    a(j+1) $\leftarrow$v
end for
```

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.

After looking at a Decrease and Conquer by a constant factor approach to solving this, we will try out the algorithm online.

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.

Example Use the application at the website

http://www.mathplayground.com/coinweighing.html

to try out the coin game. You can choose the number of coins you want to use and it counts the number of measurements you need to determine the fake coin. You do NOT know whether the fake coin is lighter or heavier so keep this in mind.

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 1/2 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 $\mathcal{O}(n \log n)$. Then to perform m searches of a sorted array of length n using Binary Search requires $m\mathcal{O}(\log n)$.

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. $(6900/993{=}6.95)$