3rd Class

Algorithms and Complexity

الخوارزميات وتعقيداتها

أستاذ المادة : م.م. منى غازي

Lecture (1)

INTRODUCTION

1. ***Introduction***

When one writes a computer program, one is generally implementing a method of solving a problem which has been previously devised. This method is often independent of the particular computer to be used: it's likely to be equally appropriate for many computers. In any case, it is the method, not the computer program itself, which must be studied to learn how the problem is being attacked. The term algorithm is universally used in computer science to describe problem-solving methods suitable for implementation as computer programs. Algorithms are the "stuff" of computer science: they are central objects of study in many, if not most, areas of the field.

Most algorithms of interest involve complicated methods of organizing the data involved in the computation. Objects created in this way are called *data structures*, and they are also central objects of study in computer science.

When a very large computer program is to be developed, a great deal of effort must go into understanding and defining the problem to be solved, managing its complexity, and decomposing it into smaller subtasks which can be easily implemented. It is often true that many of the algorithms required after the decomposition are trivial to implement. However, in most cases there are a few algorithms the choice of which is critical since most of the system resources will be spent running those algorithms.

The sharing of programs in computer systems is becoming more widespread, so that while it is true that a serious computer user will use a large fraction of the

algorithms. However, implementing simple versions of basic algorithms helps us to understand them better and thus use advanced versions more effectively in the future.

Also, mechanisms for sharing software on many computer systems often make it difficult to tailor standard programs to perform effectively on specific tasks, so that the opportunity to reimplement basic algorithms frequently arises.

Computer programs are often over optimized. It may be worthwhile to take pains to ensure that an implementation is the most efficient possible only if an algorithm is to be used for a very large task or is to be used many times.

In most situations, a careful, relatively simple implementation will suffice: the programmer can have some confidence that it will work, and it is likely to run only five or ten times slower than the best possible version, which means that it may run for perhaps an extra fraction of a second. By contrast, the proper choice of algorithm in the first place can make a difference of a factor of a hundred or a thousand or more, which translates to minutes, hours, days or more in running time.

Often several different algorithms (or implementations) are available to solve the same problem. The choice of the very best algorithm for a particular task can be a very complicated process, often involving sophisticated mathematical analysis. The branch of computer science where such questions are studied is called analysis of algorithms. Many of the algorithms that we will study have been shown to have very good performance through analysis, while others are simply known to work well through experience. We will not dwell on comparative performance issues: our goal is to learn some reasonable algorithms for important tasks. But we will try to be aware of roughly how well these algorithms might be expected to perform.

2. ***The Role of Algorithms in Computing***

What are algorithms? Why is the study of algorithms worthwhile? What is the role of algorithms relative to other technologies used in computers?

Informally, an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*. An ***algorithm*** is thus a sequence of computational steps that transform the input into the output. We can also view an ***algorithm*** as a tool for solving a well-specified computational problem.

The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

For example, one might need to sort a sequence of numbers into increasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the ***sorting problem***:

• **Input:** A sequence of *n* numbers (*a*1, *a*2, ..., *an*).

• **Output:** A permutation (reordering) (*a*1', *a*2', ..., *an'*) of the input sequence such that a1'≤a2'≤…≤an'.

For example, given the input sequence (31, 41, 59, 26, 41, 58), a sorting algorithm returns as output the sequence (26, 31, 41, 41, 58, 59). Such an input sequence is called an ***instance*** of the sorting problem. In general, an ***instance of a problem*** consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Sorting is a fundamental operation in computer science (many programs use it as an intermediate step), and as a result a large number of good sorting algorithms have been developed. Which algorithm is best for a given application depends on-among other factors the number of items to be sorted, the extent to

which the items are already somewhat sorted, possible restrictions on the item values, and the kind of storage device to be used: main memory, disks, or tapes.

An algorithm is said to be ***correct*** if, for every input instance, it halts with the correct output.

We say that a correct algorithm ***solves*** the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an answer other than the desired one. Contrary to what one might expect, incorrect algorithms can sometimes be useful, if their error rate can be controlled.

3. Algorithm Properties

- An algorithm possesses the following properties:
  - It must be correct.
  - It must be composed of a series of concrete steps.
  - There can be no ambiguity as to which step will be performed next.
  - It must be composed of a finite number of steps.
  - It must terminate.
- A ***computer program*** is an instance, or concrete representation, for an algorithm in some programming language.

4. What kinds of problems are solved by algorithms?

Practical applications of algorithms are found and include the following examples:

• The Human Genome Project has the goals of identifying all the 100,000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis. Each of these steps requires sophisticated algorithms. The saving

are in time, both human and machine, and in money, as more information can be extracted from laboratory techniques.

• The Internet enables people all around the world to quickly access and retrieve large amounts of information. In order to do so, clever algorithms are employed to manage and manipulate this large volume of data. Examples of problems which must be solved include finding good routes on which the data will travel, and using a search engine to quickly find pages on which particular information resides.

• Electronic commerce enables goods and services to be negotiated and exchanged electronically. The ability to keep information such as credit card numbers, passwords, and bank statements private is essential if electronic commerce is to be used widely. Public-key cryptography and digital signatures are among the core technologies used and are based on numerical algorithms and number theory.

• In manufacturing and other commercial settings, it is often important to allocate scarce resources in the most beneficial way. An oil company may wish to know where to place its wells in order to maximize its expected profit. An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively.

5. Easy and Hard Problems

We argue that the class of problems that can be solved in polynomial time (denoted by P) corresponds well with what we can feasibly compute. But sometimes it is difficult to tell when a particular problem is in P or not.

Theoreticians spend a good deal of time trying to determine whether particular problems are in P. To demonstrate how difficult it can be. To make this determination, we will survey a number of problems, some of which are known to be in P, and some of which we think are (probably) not in P. The difference between the two types of problem can be surprisingly small. Throughout the following, an ***"easy"*** problem is one that is solvable in polynomial time; while a ***"hard***" problem is one that we think cannot be solved in polynomial time. When we say that a problem is hard, it means that **some** instances of the problem **are hard**. It does **not** mean that **all** problem instances **are hard**.

Examples:

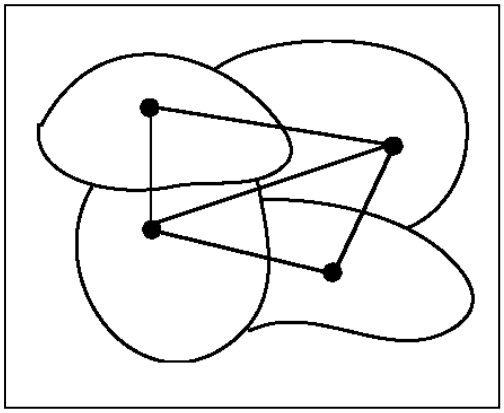
1- Eulerian Tour vs. Hamiltonian Tour

- **Eulerian Tours (Easy)**
  - INPUT: A graph G = (V, E).
  - DECIDE: Is there a path that crosses every edge exactly once and returns to its starting point?
- **Hamiltonian Tours (Hard)**
  - INPUT: A graph G = (V, E).
  - DECIDE: Is there a path that visits every vertex exactly once and returns to its starting point?

Some Facts

- **Eulerian Tours**
  - A famous mathematical theorem comes to our rescue. If the graph is connected and every vertex has even degree, then the graph is guaranteed to have such a tour. The algorithm to find the tour is a little trickier, but still doable in polynomial time.
- **Hamiltonian Tours**
  - No one knows how to solve this problem in polynomial time. The subtle distinction between visiting edges and visiting vertices changes an easy problem into a hard one.

2- <u>Map Colorability</u>



- **Map 2-colorability (Easy)**
  - INPUT: A graph G=(V, E).
  - DECIDE: Can this map be Colored with 2 colors so that no two adjacent countries have the same color?
- **Map 3-colorability (Hard)**
  - INPUT: A graph G=(V, E).
  - DECIDE: Can this map be colored with 3 colors so that no two adjacent countries have the same color?
- **Map 4-colorability (Easy)**
  - INPUT: A graph G=(V, E).
  - DECIDE: Can this map be colored with 4 colors so that no two adjacent countries have the same color?

Some Facts

- **Map 2-colorability**
  - To solve this problem, we simply color the first country arbitrarily. This forces the colors of neighboring countries to be the other color, which in turn forces the color of the countries neighboring those countries, and so on. If we reach a country which borders two countries of different color, we will know that the map cannot be two-colored; otherwise, we will produce a two coloring. So this problem is easily solvable in polynomial time.
- **Map 3-colorability**
  - This problem seems very similar to the problem above, however, it turns out to be much harder. No one knows how this problem can be solved in polynomial time. (In fact this problem is NP-complete.)
- **Map 4-colorability**.
  - Here we have an easy problem again. By a famous theorem, any map can be four-colored. It turns out that finding such a coloring is not that difficult either.

3- <u>Longest Path vs. Shortest Path</u>

- **Longest Path (Hard)**
  - INPUT: A graph G = (V, E), two vertices u, v of V, and a weighting function on E.
  - OUTPUT: The longest path between u and v.

  No one is able to come up with a polynomial time algorithm yet.

- **Shortest Path (Easy)**
  - INPUT: A graph G = (V, E), two vertices u, v of V, and a weighting function on E.
  - OUTPUT: The shortest path between u and v.

  A greedy method will solve this problem easily

4- <u>Multiplication vs. Factoring</u>

- **Multiplication (Esay)**
  - INPUT: Integers x,y.
  - OUTPUT: The product x×y.
- **Factoring (Un-multiplying) (Hard)**
  - INPUT: An integer n.
  - OUTPUT: If n is not prime, output two integers x, y such that $1 < x, y < n$ and $x \times y = n$.

Again, the problem of factoring is not known to be in P. In this case, the hardness of a problem turns out to be useful. Some cryptographic algorithms depend on the assumption that factoring is hard to ensure that a code cannot be broken by a computer.

6. <u>Hard problems</u>

Our usual measure of efficiency is speed, i.e., how long an algorithm takes to produce its result. There are some problems, however, for which no efficient solution is known. Studies an interesting subset of these problems, which are known as NP-complete.

Why are NP-complete problems interesting? First, although no efficient algorithm for an NP complete problem has ever been found, nobody has ever

proven that an efficient algorithm for one cannot exist. In other words, it is unknown whether or not efficient algorithms exist for NP-complete problems. Second, the set of NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exists for all of them. This relationship among the NP-complete problems makes the lack of efficient solutions all the more tantalizing. Third, several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms. A small change to the problem statement can cause a big change to the efficiency of the best known algorithm.

It is valuable to know about NP-complete problems because some of them arise surprisingly often in real applications.

As a concrete example, consider a trucking company with a central warehouse. Each day, it loads up the truck at the warehouse and sends it around to several locations to make deliveries. At the end of the day, the truck must end up back at the warehouse so that it is ready to be loaded for the next day. To reduce costs, the company wants to select an order of delivery stops that yields the lowest overall distance traveled by the truck. This problem is the well-known "traveling-salesman problem," and it is NP-complete.

Example:

Algorithm *fibonacci* (n):

Input: a nonnegative integer n.

Output: fib, the nth term of the fibonacci sequence.

1. **if** $n \leq 1$ **then**
2. $fib = n$
3. $\boldsymbol{else}$
4. $f1 = 0$
5. $f2 = 1$
6. $\boldsymbol{for}\ i = 2\ to\ n$
7. $fib = f1 + f2$
8. $f1 = f2$
9. $f2 = fib$
10. $\boldsymbol{end\ for}$
11. $\boldsymbol{end\ if}$
12. $\boldsymbol{return}\ fib$

Lecture (2)

ASYMPTOTIC NOTATIONS

1. Algorithmic Complexity

Algorithmic complexity is concerned about how fast or slow particular algorithm performs. We define ***complexity*** as a numerical function *T(n)* - time versus the input size *n*. We want to define time taken by an algorithm without depending on the implementation details. But you agree that *T(n)* does depend on the implementation. A given algorithm will take different amounts of time on the same inputs depending on such factors as: processor speed; instruction set, disk speed, brand of compiler and etc. The way around is to estimate efficiency of each algorithm *asymptotically*. We will measure time *T(n)* as the number of elementary "steps" (defined in any way), provided each such step takes constant time.

Let us consider two classical examples: addition of two integers. We will add two integers digit by digit (or bit by bit), and this will define a "step" in our computational model. Therefore, we say that addition of two n-bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where c is time taken by addition of two bits. On different computers, addition of two bits might take different time, say c_1 and c_2, thus the addition of two n-bit integers takes $T(n) = c_1 * n$ and $T(n) = c_2 * n$ respectively. This shows that different machines result in different slopes, but time *T(n)* grows linearly as input size increases.

The process of abstracting away details and determining the rate of resource usage in terms of the input size is one of the fundamental ideas in computer science.

2. ***Asymptotic Notations***

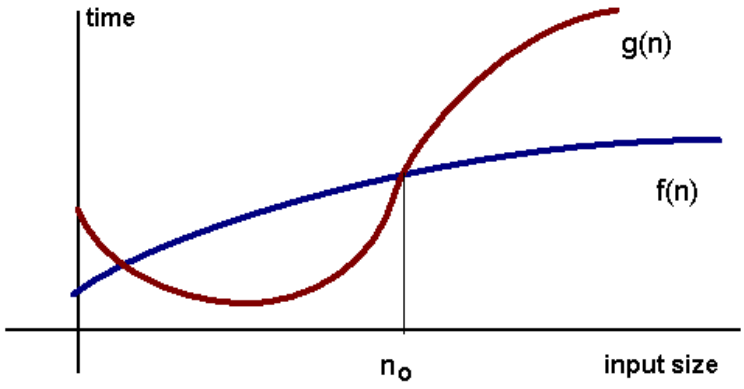
The goal of computational complexity is to classify algorithms according to their performances. We will represent the time function T(n) using the "big-O" notation to express an algorithm runtime complexity. For example, the following statement T (n) = O (n^2) says that an algorithm has a quadratic time complexity.

3. ***Definition of "big O"***

For any monotonic functions f(n) and g(n) from the positive integers to the positive integers, we say that f(n) = O(g(n)) when there exist constants c > 0 and n_0 > 0 such that :

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$

Intuitively, this means that function f(n) does not grow faster than g(n), or that function g(n) is an **upper bound** for f(n), for all sufficiently large $n \to \infty$. Here is a graphic representation of f(n) = O(g(n)) relation:



Examples:

- 1 = O(n)
- n = O(n^2)
- log(n) = O(n)
- 2 n + 1 = O(n)

The "big-O" notation is not symmetric: n = O(n^2) but $n^2 \neq$ O(n).

Standard Method to Prove Big-Oh :

1. Choose n_0= 1.
2. Assuming n > 1, find/derive a C such that

$$\frac{f(n)}{g(n)} \leq \frac{c\ g(n)}{g(n)} = c$$

This shows that $n > 1$ implies $f(n) \leq C\ g(n)$.

Keep in mind:
• $n > 1$ implies $1 < n, n < n^2, n^2 < n^3, \ldots$
• "Increase" numerator to "simplify" fraction.

Exercise: Let us prove $n^2 + 2n + 1 = O(n^2)$.

Choose n_0= 1.
Assuming $n > 1$, then

$$\frac{f(n)}{g(n)} = \frac{n^2 + 2n + 1}{n^2} \leq \frac{n^2 + 2n^2 + n^2}{n^2} = 4$$

Choose C =4. Note that $2n < 2n^2$ *and* $1 < n^2$.

Thus, $n^2 + 2n + 1$ is $O(n^2)$ because

$n^2 + 2n + 1 \leq 4n^2$ whenever $n > 1$.

H.W: Prove that 3 n + 7 = O(n)

Constant Time: O(1)

An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size. Examples:

- array: accessing any element
- fixed-size stack: push and pop methods
- fixed-size queue: enqueue and dequeue methods

Linear Time: O(n)

An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases. Examples:

- array: linear search, traversing, find minimum
- ArrayList: contains method
- queue: contains method

Logarithmic Time: O(log n)

An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size. Example:

- binary search

Recall the "twenty questions" game - the task is to guess the value of a hidden number in an interval. Each time you make a guess, you are told whether your guess is too high or too low. Twenty questions game implies a strategy that uses your guess number to halve the interval size. This is an example of the general problem-solving method known as **binary search**.

Locate the element **a** in a sorted (in ascending order) array by first comparing **a** with the middle element and then (if they are not equal) dividing the array into two sub arrays; if **a** is less than the middle element you repeat the whole procedure in the left sub array, otherwise in the right sub array. The procedure repeats until **a** is found or sub array is a zero dimension.

Note, $\log(n) < n$, when $n \to \infty$. Algorithms that run in O(log n) does not use the whole input.

Quadratic Time: $O(n^2)$

An algorithm is said to run in quadratic time if its time execution is proportional to the square of the input size. Examples:

- bubble sort, selection sort, insertion sort

4. Definition of "big Omega"

To describe lower bounds we use the big-omega notation $f(n)=\Omega(g(n))$ usually defined by saying for some constant $c>0$ and all large enough n, $f(n) \geq c\ g(n)$. This has a nice symmetry property, $f(n)=O(g(n))$ if $g(n)=\Omega(f(n))$.

Examples

- $n = \Omega(1)$
- $n^2 = \Omega(n)$
- $n^2 = \Omega(n \log(n))$
- $2\ n + 1 = \Omega\ (n)$

5. ***Definition of "big Theta"***

To measure the complexity of a particular algorithm, means to find the upper and lower bounds. A new notation is used in this case. We say that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Examples

- $2\ n = \Theta(n)$
- $n^2 + 2\ n + 1 = \Theta(\ n^2)$

6. ***Analysis of Algorithms***

The term analysis of algorithms is used to describe approaches to the study of the performance of algorithms. In this course we will perform the following types of analysis:

- the *worst-case runtime complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size a.
- the *best-case runtime complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size a.
- the *average case runtime complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size a.
- the *amortized runtime complexity* of the algorithm is the function defined by a sequence of operations applied to the input of size a and averaged over time.

Example. Let us consider an algorithm of sequential searching in an array of size n.

Its *worst-case runtime complexity* is O(n)
Its *best-case runtime complexity* is O(1)
Its *average case runtime complexity* is O(n/2)=O(n)

7. Amortized Time Complexity

Consider a dynamic array stack. In this model push() will double up the array size if there is no enough space. Since copying arrays cannot be performed in constant time, we say that push is also cannot be done in constant time. In this section, we will show that push() takes amortized constant time. Let us count the number of copying operations needed to do a sequence of pushes.

push()	**copy**	**old array size**	**new array size**
1	0	1	-
2	1	1	2
3	2	2	4
4	0	4	-
5	4	4	8
6	0	8	-
7	0	8	-
8	0	8	-
9	8	8	16

We see that 3 pushes requires 2 + 1 = 3 copies.

We see that 5 pushes requires 4 + 2 + 1 = 7 copies.

We see that 9 pushes requires 8 + 4 + 2 + 1 = 15 copies.

In general, 2^n+1 pushes requires $2^n + 2^{n-1}$+ ... + 2 + 1 = 2^{n+1} - 1 copies.

We say that the algorithm runs at **amortized constant time**.

Lecture (3)

COMPLEXITY EXAMPLES

1. Running Time Functions

Most algorithms have a primary parameter N, usually the number of data items to be processed, which affects the running time most significantly. The parameter N might be the degree of a polynomial, the size of a file to be sorted or searched, the number of nodes in a graph, etc. proportional to one of the following functions:

1. **(1)** Most instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is constant. This is obviously the situation to strive for algorithm design.
2. **(log N)** When the running time of a program is logarithmic, the program gets slightly slower as N grows. This running time commonly occurs in programs which solve a big problem by transforming it into a smaller problem by cutting the size by some constant fraction. For our range of interest, the running time can be considered to be less than a **large** constant. The base of the logarithm changes the constant, but not by much: when N is a thousand, log N is 3 if the base is 10, 10 if the base is 2; when N is a million, 1og N is twice as great. Whenever N doubles, log N increases by a constant, but log N doesn't double until N increases to N^2.
3. (**N)** When the running time of a program is linear, it generally is the case that a small amount of processing is done on each input

element. When N is a million, then so is the running time. Whenever N doubles, then so does the running time. This is the optimal situation for an algorithm that must process N inputs (or produce N outputs).

4. (N **log N)** This running time arises in algorithms which solve a problem by breaking it up into smaller sub problems, solving them independently, and then combining the solutions. For lack of a better adjective (linearithmic), we'll say that the running time of such an algorithm is "N log N." When N is a million, N log N is perhaps twenty million. When N doubles, the running times more than doubles (but not much more).

5. ($\mathbf{N^2}$ **)** When the running time of an algorithm is *quadratic,* it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms which process all pairs of data items (perhaps in a double nested loop). When N is a thousand, the running time is a million. Whenever N doubles, the running time increases fourfold.

6. **($\mathbf{N^3}$)** Similarly, an algorithm which processes triples of data items (perhaps in a triple-nested loop) has a *cubic* running time and is practical for use only on small problems. When N is a hundred, the running time is a million. Whenever N doubles, the running time increases eightfold.

7. **($\mathbf{2^N}$)** Few algorithms with *exponential* running time are likely to be appropriate for practical use, though such algorithms arise naturally as "brute-force" solutions to problems. When N is twenty, the running time is a million. Whenever N doubles, the running time squares.

The O - Examples

$f(n) = 2n + 3$; $f(n) = O(n)$

$f(n) = 6n^2 + 235$; $f(n) = O(n^2)$

$f(n) = 6n + 567\ln(n)$; $f(n) = O(n)$

$f(n) = 6n \times \ln(n)$; $f(n) = O(n)$

$f(n) = 2\exp(n) + n\ln(n) + 456n$; $f(n) = O(\exp(n))$

A few other functions do arise. For example, an algorithm with N^2 inputs that has a running time that is cubic in N is more properly classed as an $N^{3/2}$ algorithm. Also some algorithms have two stages of sub problem decomposition, which leads to a running time proportional to $N(\log N)^2$. Both of these functions should be considered to be much closer to N log N than to N^2 for large N.

One further note on the "log" function. The base of the logarithm changes things only by a constant factor. Since we usually deal with analytic results only to within a constant factor, it doesn't matter much what the base is, so we refer to "log N," etc. On the other hand, it is sometimes the case that concepts can be explained more clearly when some specific base is used. In mathematics, the natural logarithm (base e = 2.718281828.. .) arises so frequently that a special abbreviation is commonly used: log N = Ln N.

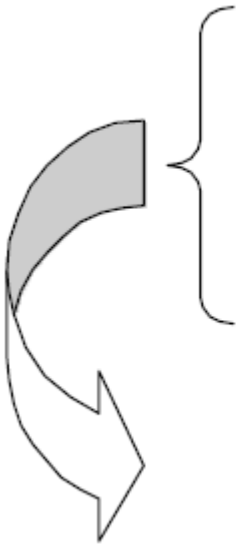
If a function (which describes the order of growth of an algorithm) is a *sum* of several terms, its order of growth is determined by the fastest growing term. In particular, if we have a polynomial

$$\mathbf{p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0}$$

its growth is of the order n^k:

$$\mathbf{p(n) = O(n^k)}$$

Example:



```
{perform any statement S1}                          O(1)
for (i=0; i < n; i++)
{
        {perform any statement(s) S2}               O(n)

        {run through another loop n times}          O(n^2)
}
```

Total Execution Time: $1 + n + n^2$ therefore, $\mathbf{O(n^2)}$

Space complexity

The (space) complexity of a program (for a given input) is the number of elementary objects that this program needs to store during its execution. This number is computed with respect to the size n of the input data.

Note:

We thus make the assumption that each elementary object needs the same amount of space.

Space complexity is measured by using polynomial amounts of memory, with an infinite amount of time.

The difference between space complexity and time complexity is that space can be reused. Space complexity is not affected by determinism or non determinism.

Amount of computer memory required during the program execution, as a function of the input size

How to Determine Complexities

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

1. Sequence of statements

```
statement 1;
statement 2;
  ...
statement k;
```

(Note: this is code that really is exactly k statements; this is **not** an unrolled loop like the N calls to *add* shown above.) The total time is found by adding the times for all statements:

Total time = time (statement 1) + time (statement 2) + ... + time (statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also

constant: O(1). In the following examples, assume the statements are simple unless noted otherwise.

2- if-then-else statements

```
if (condition) {
 Sequence of statements 1
     }
  else {
 sequence of statements 2
}
```

Here, either sequence 1 will execute, or sequence 2 will execute. Therefore, the worst-case time is the slowest of the two possibilities: max (time (sequence 1), time (sequence 2)). For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be O(N).

3- for loops

```
for (i = 0; i < N; i++) {
 sequence of statements
}
```

The loop executes N times, so the sequence of statements also executes N times. Since we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

4- Nested loops

First we'll consider loops where the number of iterations of the inner loop is independent of the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        sequence of statements
    }
}
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M). In a common special case where the stopping condition of the inner loop is j < N instead of j < M (i.e., the inner loop also executes N times), the total complexity for the two loops is $O(N^2)$.

Now let's consider nested loops where the number of iterations of the inner loop depends on the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
    for (j = i+1; j < N; j++) {
        sequence of statements
    }
}
```

Now we can't just multiply the number of iterations of the outer loop times the number of iterations of the inner loop, because the inner loop has a different number of iterations each time. So let's think about how many iterations that inner loop has. That information is given in the following table:

Value of i	Number of iterations of inner loop
0	N
1	N-1
2	N-2
...	...
N-2	2
N-1	1

So we can see that the total number of times the sequence of statements executes is: N + N-1 + N-2 + ... + 3 + 2 + 1. We've seen that formula before: the total is $O(N^2)$.

5- Statements with method calls:

When a statement involves a method call, the complexity of the statement includes the complexity of the method call. Assume that you know that method *f* takes constant time, and that method *g* takes time proportional to (linear in) the value of its parameter *k*. Then the statements below have the time complexities indicated.

```
f(k);  // O(1)
g(k);  // O(k)
```

When a loop is involved, the same rule applies. For example:

```
for (j = 0; j < N; j++) g(N);
```

has complexity (N^2). The loop executes N times and each method call g(N) is complexity O(N).

Exercises:

What is the worst-case complexity of the each of the following code fragments?

1. Two loops in a row:

```
for (i = 0; i < N; i++) {
  sequence of statements
    }
   for (j = 0; j < M; j++) {
   sequence of statements
  }
```

How would the complexity change if the second loop went to N instead of M?

2. A nested loop followed by a non-nested loop:

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
     sequence of statements
                          }
                           }
for (k = 0; k < N; k++) {
sequence of statements
                          }
```

3. A nested loop in which the number of times the inner loop executes depends on the value of the outer loop index:

```
for (i = 0; i < N; i++) {
  for (j = N; j > i; j--) {
     sequence of statements
                          }
                         }
```

Solution

1. The first loop is O(N) and the second loop is O(M). Since you don't know which is bigger, you say this is O(N+M). This can also be written as O(max(N,M)). In the case where the second loop goes to N instead of M the complexity is O(N). You can see this from either expression above. O(N+M) becomes O(2N) and when you drop the constant it is O(N). O(max(N,M)) becomes O(max(N,N)) which is O(N).
2. The first set of nested loops is $O(N^2)$ and the second loop is O(N). This is $O(\max(N^2,N))$ which is $O(N^2)$.
3. This is very similar to our earlier example of a nested loop where the number of iterations of the inner loop depends on the value of the index of the outer loop. The only difference is that in this example the inner-loop index is counting down from N to i+1. It is still the case that the inner loop executes N times, then N-1, then N-2, etc, so the total number of times the innermost "sequence of statements" executes is $O(N^2)$.

Exercise

For each of the following loops with a method call, determine the overall complexity. As above, assume that method *f* takes constant time, and that method *g* takes time linear in the value of its parameter.

1. for (j = 0; j < N; j++) f(j);
2. for (j = 0; j < N; j++) g(j);
3. for (j = 0; j < N; j++) g(k);

Solution

1. Each call to f(j) is O(1). The loop executes N times so it is N *O(1) or O(N).
2. The first time the loop executes j is 0 and g(0) takes "no operations". The next time j is 1 and g(1) takes 1 operations. The last time the loop executes j is N-1 and g(N-1) takes N-1 operations. The total work is the sum of the first N-1 numbers and is $O(N^2)$.
3. Each time through the loop g(k) takes k operations and the loop executes N times. Since you don't know the relative size of k and N, the overall complexity is O(N * k).

Lecture (4)

ALGORITHM TYPES AND CLASSIFICATIONS

The speed of an algorithm is measured in terms of number of basic operations it performs. Consider an algorithm that takes N as input and performs various operations. The correlation between number of operations performed and time taken to complete is as follows (consider N as 1,000 and speed of processor as 1 Ghz).

Problem whose running time does not depend on input size– constant time. (very small).

N operations — 1 ns

N*log N operations —- 20 ns

N^2 operations——- 1ms

2^N operations —— 10^{1224} seconds

N! operations —- Unimaginable time.

Different types of algorithms

Every algorithm falls under a certain class. Basically they are

1) Brute force

2) Divide and conquer

3) Decrease and conquer

4) Dynamic programming

5) Greedy algorithm

6) Transform and conquer

7) Backtracking algorithm and so on.

Brute force algorithm

Brute force implies using the definition to solve the problem in a straightforward manner. Brute force algorithms are usually the easiest to implement, but the disadvantage of solving a problem by brute force is that it is usually very slow and can be applied only to problems where input size is small.

Divide and conquer algorithm

In divide and conquer method, we divide the size of a problem by a constant factor in each iteration. This means we have to process lesser and lesser part of the original problem in each iteration. Some of the fastest algorithms belong to this class. Divide and conquer algorithms have logarithmic runtime.

Decrease and conquer algorithm

This kind of problem is same as divide and conquer, except, here we are decreasing the problem in each iteration by a constant size instead of constant factor.

Dynamic programming

The word ‘dynamic’ refers to the method in which the algorithm computes the result. Sometimes, a solution to the given instance of problem depends on the solution to smaller instance of sub-problems. It exhibits the property of overlapping sub-problems. Hence, to solve a problem we may have to recompute same values again and again for smaller sub-problems. Hence, computing cycles are wasted.

To remedy this, we can use dynamic programming technique. Basically, in dynamic programming, we “remember” the result of each sub-problem. Whenever we need it, we will use that value instead of recomputing it again and again.

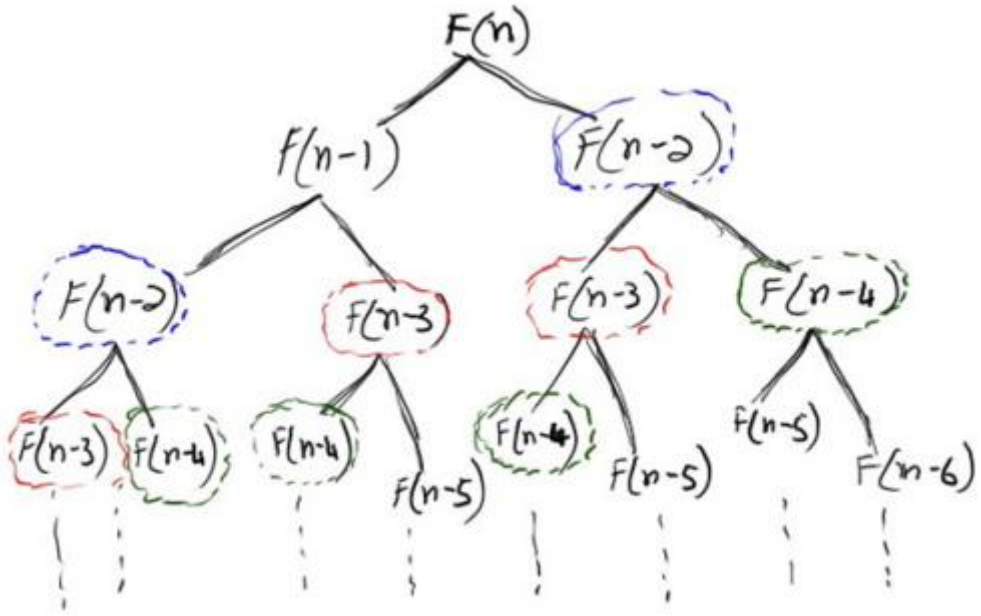
Here, we are trading space for time. i.e. – we are using more space to hold the computed values to increase the execution speed drastically.

A good example for a problem that has overlapping sub-problem is the relation for Nth Fibonacci number.

It is defined as F(n)= F(n-1) + F (n-2) .

Note that the Nth Fibonacci number depends on previous two Fibonacci number.

If we compute F(n) in conventional way, we have to calculate in following manner



The similar colored values are those that will be calculated again and again. Note that F(n-2) is computed 2 times, F(n-3) 3 times and so on … Hence, we are wasting a lot of time. In fact this recursion will perform 2^N operations for a given N, and it is not at all solvable for N>40 on a modern PC within at least a year.

The solution to this is to store each value as we compute it and retrieve it directly instead of re calculating it. This transforms the exponential time algorithm into a linear time algorithm.

Hence, dynamic programming is a very important technique to speed up the problems that have overlapping sub problems.

Greedy algorithm

For many problems, making greedy choices leads to an optimal solution. These algorithms are applicable to optimization problems.

In a greedy algorithm, in each step, we will make a locally optimum solution such that it will lead to a globally optimal solution. Once a choice is made, we cannot retract it in later stages.

Proving the correctness of a greedy algorithm is very important, since not all greedy algorithms lead to globally optimum solution.

For ex- consider the problem where you are given coins of certain denomination and asked to construct certain amount of money in minimum number of coins.

Let the coins be of 1, 5, 10, 20 cents

If we want change for 36 cents, we select the largest possible coin first (greedy choice).

According to this process, we select the coins as follows-

20

20 + 10

20 + 10 + 5

20 + 10 + 5 + 1 = 36.

For coins of given denomination, the greedy algorithm always works.

But in general this is not true.

Consider the denomination as 1, 3, 4 cents

To make 6 cents, according to greedy algorithm the selected coins are 4 + 1 + 1

But, the minimum coins needed are only 2 (3 + 3)

Hence, greedy algorithm is not the correct approach to solve the ‘change making’ problem.

In fact, we can use dynamic programming to arrive at optimal solution to this problem.

Transform and conquer algorithm

Sometimes it is very hard or not so apparent as to how to arrive at a solution for a particular problem.

In this case, it is easier to transform the problem into something that we recognize, and then try to solve that problem to arrive at the solution.

Consider the problem of finding LCM (least common multiple) of a number. Brute force approach of trying every number and seeing if it is the LCM is not the best approach. Instead, we can find the GCD (greater common divisor) of the problem using a very fast algorithm known as Euclid’s algorithm and then use that result to find the LCM as LCM (a , b) = (a * b) / GCD (a , b).

Backtracking algorithm

Backtracking approach is very similar to brute force approach. But the difference between backtracking and brute force is that, in brute force approach, we are generating every possible combination of solution and testing if it is a valid solution. Whereas, in backtracking, each time you generate a solution, you are testing if it satisfies all condition, and only then we continue generating subsequent solutions, else we will backtrack and go on a different path of finding solution.

A famous example to this problem is the N Queens problem. According to the problem, we are given a N X N sized chessboard. We have to place N queens on the chessboard such that no queens are under attack from any other queen.

We proceed by placing a queen in every column and appropriate row. Every time we place a queen, we check whether it is under attack. If so, then we will choose a different cell under that column. You can visualize the process like a tree. Each node in the tree is a chessboard of different configuration. At any stage if we are unable to proceed, then we backtrack from that node and proceed by expanding other nodes.

An advantage of this method over brute force is that the numbers of candidates generated are very less compared to brute force approach. Hence we can isolate valid solutions quickly.

Ex- for an 8 X 8 chess board, if we follow brute force approach, we have to generate 4,426,165,368 solutions and test each of them. Whereas, in backtracking approach, it gets reduced to 40,320 solutions.

Q/ write a c++ program to print a pyramid of digits as shown below for n number of lines.

```
        1
      2 3 2
    3 4 5 4 3
  4 5 6 7 6 5 4
```

```
/*program to print digits in pyramidal form*/
#include <iostream.h>
void main()
{
int i,j,k,r,m,n;
cout<<"\n how many lines?";
cin>>n;

/*loop 1 to print n lines*/
for(r=1;r<=n;r++)
{
```

```
/*loop 2 to print spaces*/
for(i=1;i<=n-r;i++)
cout<<" ";

/*loop to print digits in the left side of axis*/
m=r;
for(j=1;j<=r;j++)
{
cout<<m;
m++;
}

/*loop to print digits rightside of axis*/
If (r>1 )
{
m=m-2;
for(k=1;k<r-1;k++)
{
cout<<m;
m--;
}
}

Cout<<endl;
}
}
```

H.W.: Write algorithm for this program.

Lecture (5)

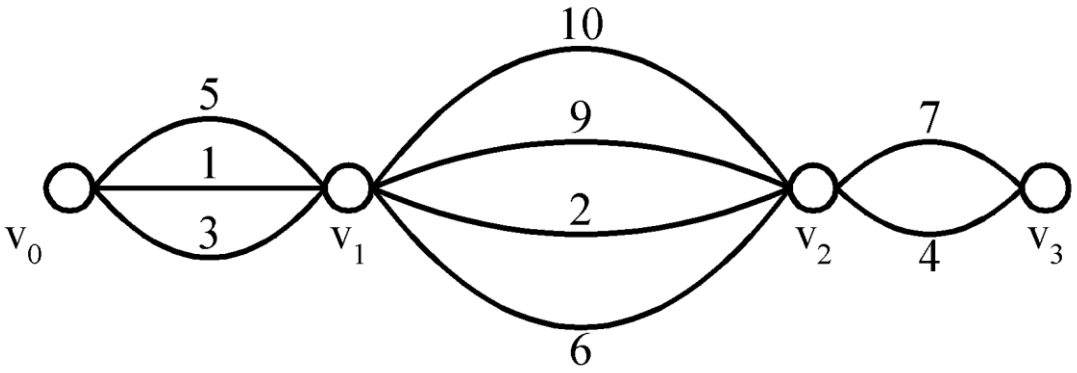
GREEDY ALGORITHM

Suppose that a problem can be solved by a sequence of decisions. The greedy method has that each decision is locally optimal. These locally optimal solutions will finally add up to a globally optimal solution.

Only a few optimization problems can be solved by the greedy method.

Shortest paths on a special graph

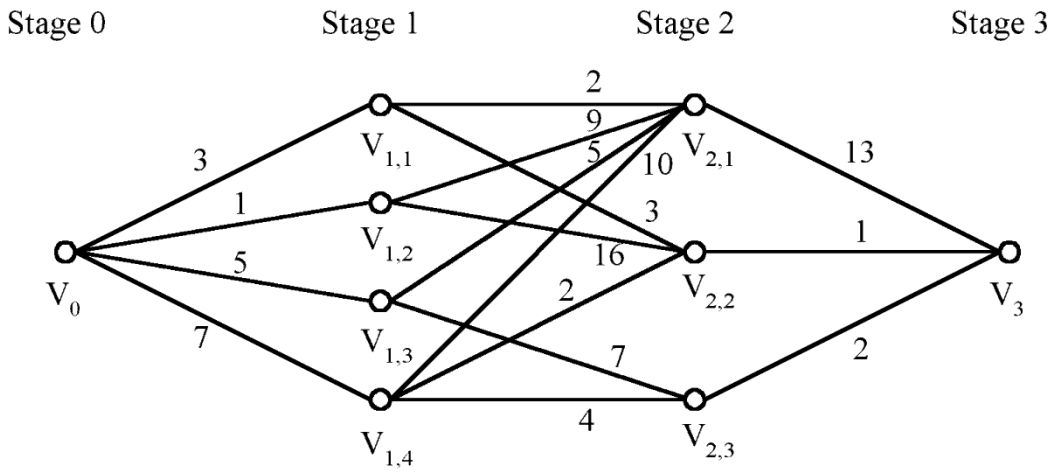
- Problem: Find a shortest path from v_0 to v_3.



- The greedy method can solve this problem.
- The shortest path: 1 + 2 + 4 = 7.

Shortest paths on a multi-stage graph

- Problem: Find a shortest path from v_0 to v_3 in the multi-stage graph.



- Greedy method: $v_0v_{1,2}v_{2,1}v_3 = 23$
- Optimal: $v_0v_{1,1}v_{2,2}v_3 = 7$

The greedy method does not work.

Solution of the above problem

- $d_{min}(i,j)$: minimum distance between i and j.

$$d_{min}(v_0,v_3)=\min\begin{cases} 3+d_{min}(v_{1,1},v_3) \\ 1+d_{min}(v_{1,2},v_3) \\ 5+d_{min}(v_{1,3},v_3) \\ 7+d_{min}(v_{1,4},v_3) \end{cases}$$

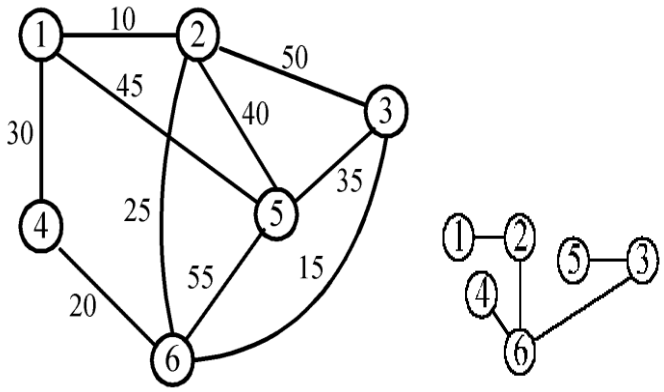
- This problem can be solved by the dynamic programming method.

Minimum spanning trees (MST)

- It may be defined on Euclidean space points or on a graph.
- G = (V, E): weighted connected undirected graph
- Spanning tree : S = (V, T), T $\subseteq$ E, undirected tree
- Minimum spanning tree(MST) : a spanning tree with the smallest total weight.

An example of MST

- A graph and one of its minimum costs spanning tree



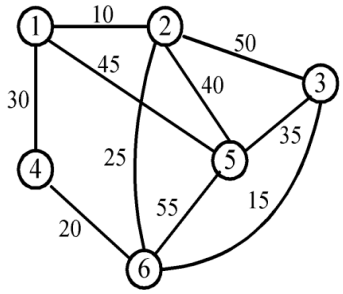
Kruskal's algorithm for finding MST:

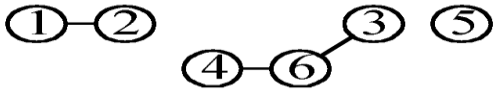
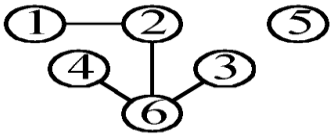
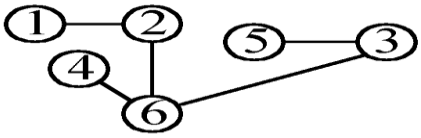
Step 1: Sort all edges into nondecreasing order.

Step 2: Add the next smallest weight edge to the forest if it will not cause a cycle.

Step 3: Stop if n-1 edges. Otherwise, go to Step2.

An example for Kruskal's algorithm:



Edge	Cost	Spanning Forest
		
(1,2)	10	
(3,6)	15	
(4,6)	20	
(2,6)	25	
(1,4)	30	(reject)
(3,5)	35	

Prim's algorithm for finding MST:

Step 1: $x \in V$, Let $A = \{x\}$, $B = V - \{x\}$.

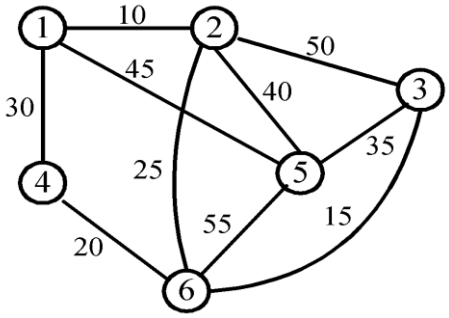
Step 2: Select $(u, v) \in E$, $u \in A$, $v \in B$ such that (u, v) has the smallest weight between A and B.

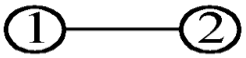
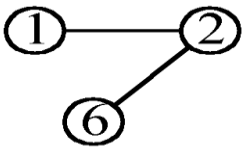
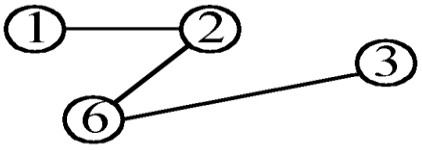
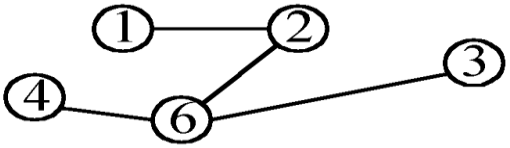
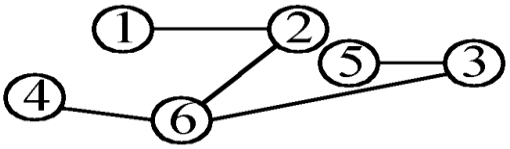
Step 3: Put (u, v) in the tree. $A = A \cup \{v\}$, $B = B - \{v\}$

Step 4: If $B = \varnothing$, stop; otherwise, go to Step 2.

Time complexity : $O(n^2)$, $n = |V|$.

An example for Prim's algorithm



Edge	Cost	Spanning tree
(1,2)	10	
(2,6)	25	
(3,6)	15	
(6,4)	20	
(3,5)	35	

What are difference between Prim's algorithm and Kruskal's algorithm for finding the minimum spanning tree of a graph :

Prim's method starts with one vertex of a graph as your tree, and adds the smallest edge that grows your tree by one more vertex. Kruskal starts with all of the vertices of a graph as a forest, and adds the smallest edge that joins two trees in the forest.

Prim's method is better when

- You can only concentrate on one tree at a time
- You can concentrate on only a few edges at a time

Kruskal's method is better when

- You can look at all of the edges at once
- You can hold all of the vertices at once
- You can hold a forest, not just one tree

Basically, Kruskal's method is more time-saving (you can order the edges by weight and burn through them fast), while Prim's method is more space-saving (you only hold one tree, and only look at edges that connect to vertices in your tree).

Lecture (6)

DIVIDE AND CONQUER ALGORITHM

The ***divide and conquer*** strategy solves a problem by :

1. Breaking into ***sub problems*** that are themselves smaller instances of the same type of problem.
2. Recursively solving these sub problems.
3. Appropriately combining their answers.

Two types of sorting algorithms which are based on this divide and conquer algorithm:

1. **Quick sort:** Quick sort also uses few comparisons. Like heap sort it can sort "in place" by moving data in an array.

2. **Merge sort:** Merge sort is good for data that's too big to have in memory at once, because its pattern of storage access is very regular. It also uses even fewer comparisons than heap sort, and is especially suited for data stored as linked lists.

Table for the running time of the algorithms :

In this table, *n* is the number of records to be sorted. The columns "Best", "Average", and "Worst" give the time complexity in each case. "Memory" denotes the amount of auxiliary storage needed beyond that used by the list itself.

Name	Best	Average	Worst	Memory	Stable	Method
Heap sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	No	Selection
Merge sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	Yes	Merging
Quick sort	O(nlogn)	O(nlogn)	O(n2)	O(logn)	No	Partitioning

Here we are discussing about the two algorithms quick sort and merge sort. We will also discuss about their ***complexity running time*** of algorithms.

Quick sort

Quick sort is one of the fastest and simplest sorting algorithms, which uses partitioning as its main idea. It works recursively by a divide-and-conquer strategy.

Example: Pivot about 10.

17 12 6 19 23 8 5 10 - before

6 8 5 10 23 19 12 17 – after

Partitioning places all the elements less than the pivot in the left part of the array, and all elements greater than the pivot in the right part of the array. The pivot fits in the slot between them.

Example: pivot about 10

| 17 12 6 19 23 8 5 | 10

| 12 6 19 23 8 5| 17

| 6 19 23 8 5 | 12 17

6 | 19 23 8 5 | 12 17

6 |23 8 5| 19 12 17

6 | 8 5 | 23 19 12 17

6 8| 5 | 23 19 12 17

6 8 5 ||23 19 12 17

6 8 5 10 23 19 12 17

Note that the pivot element ends up in the correct place in the total order.

Quick sort algorithm:

```
Algorithm quicksort(q)
var list less, pivotList, greater
if length(q) ≤ 1
return q
else
select a pivot value from q
for each x in q except the pivot element
if x < pivot then add x to less
if x ≥ pivot then add x to greater
add pivot to pivotList
return concatenate(quicksort(less), pivotList, quicksort(greater))
```

Complexity of Quicksort

Best case:

Set up a recurrence relation for T(n), the time needed to sort a list of size n. Because a single quicksort call involves O(n) work plus two recursive calls on lists of size n/2 in the best case, the relation would be:

$T(n) = O(n) + 2T(n/2)$

The master theorem tells us that $T(n) = \Theta(n \log n)$.

Average case:

The average number of comparisons over all permutations of the input sequence can be estimated accurately by solving the recurrence relation:

$$C(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(C(i) + C(n-i-1)) = 2n\ln n = 1.39n\log_2 n.$$

Here, n - 1 is the number of comparisons the partition uses. Since the pivot is equally likely to fall anywhere in the sorted list order, the sum is averaging over all possible splits.

Worst case:

In the worst case, however, the two sublists have size 1 and n-1, and the call tree becomes a linear chain of n nested calls. The recurrence relation is:

T(n) = O(n) + T(1) + T(n - 1) = O(n) + T(n - 1)

This is the same relation as for insertion sort and selection sort, and it solves to T(n) = $\Theta(n^2)$.

Merge Sort

Merge Sort is a O(nlogn) sorting algorithm. It is easy to implement merge sort such that it is stable meaning it preserves the input order of equal elements in the sorted output. It is a comparison sort.

Example:

1	2	3	4	5	6	7	8	9	10	11	12	13
A [31	23	01	17	19	28	09	03	13	15	22	08	29]

1 : 13

1 : 7 8 : 13

1 : 4 5 : 7 8 : 10 11 : 13

1 : 2 3 : 4 5 : 6 7 : 7 8 : 9 10 : 10 11 : 12 13 : 13

31 23 01 17 19 28 09 03 13 15 22 08 29

23 31 01 17 19 28 09 03 13 15 08 22 29

01 17 23 31 09 19 28 03 13 15 08 22 29

01 09 17 19 23 28 31 03 08 13 15 22 29

01 03 08 09 13 15 17 19 22 23 29 31

Merge sort algorithm:

Algorithm: mergesort (A, left, right)

Input: An array A of numbers , the bounds left and

Right for the elements to be sorted

Output: A [left…right] is sorted

Process:

```
If (left<right) {      /*we have at least two elements to sort*/
Mid= [(left +right)/2]
Mergesort (A, left, mid)        /*now A [left….mid] is sorted*/
mergesort(A,mid+1,right)    /*now A [mid+1….right] is sorted */
Merge (A, left, mid, right)   /* merge A [left ...mid] with A [mid+1...right]*/
}
```

Merging:

```
Merge(array A, int left, int mid, int right)
{
array B[left..right]
i = k = left                                    // initialize pointers
j = mid+1
while (i <= mid and j <= right)  {        // while both subarrays are nonempty
if (A[i] <= A[j])   B[k++] = A[i++]    // copy from left subarray
else                  B[k++] = A[j++]    // copy from right subarray
}
while (i <= mid)     B[k++] = A[i++]      // copy any leftover to B
while (j <= right)    B[k++] = A[j++]
for  i = left  to  right    A[i] = B[i]          // copy B back to A
}
```

Complexity of Merge Sort

Best case:

If the running time of merge sort for a list of length n is $T(n)$, then the recurrence $\boldsymbol{T(n) = 2T(n/2) + n}$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists)

Worst and Average case:

In sorting n items, merge sort has an average and worst-case performance of $\boldsymbol{O(n \log n)}$.

The worst case, merge sort does exactly $\boldsymbol{(n \log n - 2^{\log n} + 1)}$ comparisons, which is between $\boldsymbol{(n \log n - n + 1)}$ and $\boldsymbol{(n \log n - 0.9139 \cdot n + 1)}$ [logs are base 2]. Merge sort's ***worst*** case is found simultaneously with quicksort's ***best*** case.

Lecture (7)

DYNAMIC PROGRAMMING

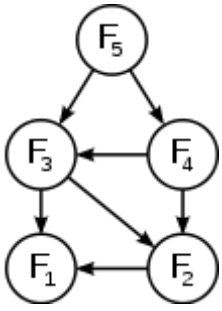
The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations: once the solution to a given subproblem has been computed, it is stored, the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems **grows exponentially** as a function of the size of the input.

There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub problems. If a problem can be solved by combining optimal solutions to *non-overlapping* sub problems, the strategy is called "divide and conquer". This is why merge sort and quick sort are not classified as dynamic programming problems.

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub problems. Consequently, the first step towards devising a dynamic programming solution is to check whether the problem exhibits such optimal substructure. Such optimal substructures are usually described by means of

recursion. For example, given a graph $G=(V,E)$, the shortest path p from a vertex u to a vertex v exhibits optimal substructure: take any intermediate vertex w on this shortest path p. If p is truly the shortest path, then the path p_1 from u to w and p_2 from w to v are indeed the shortest paths between the corresponding vertices. Hence, one can easily formulate the solution for finding shortest paths in a recursive manner.

Overlapping subproblems means that the space of subproblems must be small, that is, any recursive algorithm solving the problem should solve the same subproblems over and over, rather than generating new subproblems. For example, consider the recursive formulation for generating the Fibonacci series: $F_i = F_{i-1} + F_{i-2}$, with base case $F_1 = F_2 = 1$. Then $F_{43} = F_{42} + F_{41}$, and $F_{42} = F_{41} + F_{40}$. Now F_{41} is being solved in the recursive subtrees of both F_{43} as well as F_{42}. Even though the total number of subproblems is actually small (only 43 of them), we end up solving the same problems over and over if we adopt a naive recursive solution such as this. Dynamic programming takes account of this fact and solves each subproblem only once.



The subproblem graph for the Fibonacci sequence.

This can be achieved in either of two ways:

- *Top-down approach*: This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its subproblems, and if its subproblems are overlapping, then one can easily store the solutions to the subproblems in a table. Whenever we attempt to solve a new subproblem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the subproblem and add its solution to the table.

- *Bottom-up approach*: Once we formulate the solution to a problem recursively as in terms of its subproblems, we can try reformulating the problem in a bottom-up fashion: try solving the subproblems first and use their solutions to build-on and arrive at solutions to bigger subproblems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger subproblems by using the solutions to small subproblems. For example, if we already know the values of F_{41} and F_{40}, we can directly calculate the value of F_{42}.

In this lecture we study a powerful algorithm design technique that is widely used to solve combinatorial optimization problems. An algorithm that employs this technique is not recursive by itself, but the underlying solution of the problem is usually stated in the form of a recursive function. Unlike the case in divide-and-conquer algorithms, immediate implementation of the recurrence results in identical recursive calls that are executed more than once. For this reason, this technique resorts to evaluating the recurrence in a bottom-up manner, saving intermediate results that are used later on to compute the desired solution. This technique applies to many combinatorial optimization problems to derive efficient algorithms. It is also used to improve the time complexity of the brute-force methods to solve some of the NP-hard problems . For example, the traveling salesman problem can be solved in time $O(n^2 2^n)$ using dynamic programming, which is superior to the $\Theta(n!)$ bound of the obvious algorithm that enumerates all possible tours. The two simple examples that follow illustrate the essence of this design technique.

Example One of the most popular examples used to introduce recursion and induction is the problem of computing the Fibonacci sequence:

$$f_1 = 1, f_2 = 1, f_3 = 2, f_4 = 3, f_5 = 5, f_6 = 8, f_7 = 13, \ldots.$$

Each number in the sequence $2, 3, 5, 8, 13, \ldots$ is the the sum of the two preceding numbers. Consider the inductive definition of this sequence:

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ f(n-1) + f(n-2) & \text{if } n \geq 3. \end{cases}$$

This definition suggests a recursive procedure that looks like the following (assuming that the input is always positive).

```
1. procedure f(n)
2. if (n = 1) or (n = 2) then return 1
3. else return f(n − 1) + f(n − 2)
```

This recursive version has the advantages of being concise, easy to write and debug and, most of all, its abstraction. It turns out that there is a rich class of recursive algorithms and, in many instances, a complex algorithm can be written succinctly using recursion. We have already encountered in the previous chapters a number of efficient algorithms that possess the merits of recursion. It should not be thought, however, that the recursive procedure given above for computing the Fibonacci sequence is an efficient one. On the contrary, it is far from being efficient, as there are many duplicate recursive calls to the procedure. To see this, just expand the recurrence a few times:

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ &= 2f(n-2) + f(n-3) \\ &= 3f(n-3) + 2f(n-4) \\ &= 5f(n-4) + 3f(n-5) \end{aligned}$$

This leads to a huge number of identical calls. If we assume that computing $f(1)$ or $f(2)$ requires a unit amount of time, then the time complexity of this procedure can be stated as

$$T(n) = \begin{cases} 1 & if\ n = 1\ or\ n = 2 \\ T(n-1) + T(n-2) & if\ n \geq 3 \end{cases}$$

The Longest Common Subsequence Problem

A simple problem that illustrates the underlying principle of dynamic programming is the following problem. Given two strings A and B of lengths n and m, respectively, over an alphabet Σ, determine the *length of the longest subsequence that is common to both A and B*. Here, a subsequence of $A = a_1 a_2, \ldots a_n$ is a string of the form $a_{i_1} a_{i_2}, \ldots a_{i_k}$, where each i_j is between 1 and n and $1 \leq i_1 < i_2 < \ldots < i_k \leq n$. For example, if $\Sigma = \{x, y, z\}$, $A = zxyxyz$ and $B = xyyzx$, then xyy is a subsequence of length 3 of both A and B. However, it is not the *longest* common subsequence of A and B, since the string $xyyz$ is also a common subsequence of length 4 of both A and B. Since these two strings do not have a common subsequence of length greater than 4, the length of the longest common subsequence of A and B is 4.

One way to solve this problem is to use the brute-force method: enumerate all the 2^n subsequences of A, and for each subsequence determine if it is also a subsequence of B in $\Theta(m)$ time. Clearly, the running time of this algorithm is $\Theta(m2^n)$, which is exponential.

In order to make use of the dynamic programming technique, we first find a recursive formula for the length of the longest common subsequence. Let $A = a_1 a_2, \ldots a_n$ and $B = b_1 b_2, \ldots b_m$. Let $L[i, j]$ denote the length of a *longest* common subsequence of $a_1 a_2, \ldots a_i$ and $b_1 b_2, \ldots b_j$. Note that i or j may be zero, in which case one or both of $a_1 a_2, \ldots a_i$ and $b_1 b_2, \ldots b_j$ may be the empty string. Naturally, if $i = 0$ or $j = 0$, then $L[i, j] = 0$. The following observation is easy to prove:

+

The algorithm

Using the technique of dynamic programming to solve the longest common subsequence problem is now straightforward. We use an $(n + 1) \times (m + 1)$ table to compute the values of $L[i, j]$ for each pair of values of i and j, $0 \leq i \leq n$ and $0 \leq j \leq m$. We only need to fill the table $L[0..n, 0..m]$ row by row using the above formula. The method is formally described in Algorithm LCS.

Algorithm LCS
Input: Two strings A and B of lengths n and m, respectively, over an alphabet Σ.
Output: The length of the longest common subsequence of A and B.

1. **for** $i \leftarrow 0$ **to** n
2. $\quad L[i,0] \leftarrow 0$
3. **end for**
4. **for** $j \leftarrow 0$ **to** m
5. $\quad L[0,j] \leftarrow 0$
6. **end for**
7. **for** $i \leftarrow 1$ **to** n
8. $\quad$ **for** $j \leftarrow 1$ **to** m
9. $\quad\quad$ **if** $a_i = b_j$ **then** $L[i,j] \leftarrow L[i-1,j-1]+1$
10. $\quad\quad$ **else** $L[i,j] \leftarrow \max\{L[i,j-1], L[i-1,j]\}$
11. $\quad\quad$ **end if**
12. $\quad$ **end for**
13. **end for**
14. **return** $L[n,m]$

Example shows the result of applying Algorithm LCS on the instance A ="xyxxzxyzxy" and B ="zxzyyzxxyxxz".

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	**1**	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	2	**2**	2	2	2	2	2	2	2
3	0	0	1	1	2	2	2	**3**	3	3	3	3	3
4	0	0	1	1	2	2	2	3	4	4	**4**	4	4
5	0	1	1	2	2	2	3	3	4	4	4	4	5
6	0	1	2	2	2	2	3	4	4	4	5	**5**	5
7	0	1	2	2	3	3	3	4	4	5	5	5	5
8	0	1	2	3	3	3	4	4	4	5	5	5	**6**
9	0	1	2	3	3	3	4	5	5	5	6	6	6
10	0	1	2	3	4	4	4	5	5	6	6	6	6

First, row 0 and column 0 are initialized to 0. Next, the entries are filled row by row by executing Steps 5 and 6 exactly mn times. This generates the rest of

the table. As shown in the table, the length of a longest common subsequence is 6. One possible common subsequence is the string "xyxxxz" of length 6, which can be constructed from the entries in the table in bold face.

Lecture (8)

NETWORK FLOW PROBLEM

1. Introduction

The network flow problem is an example of a beautiful theoretical subject that has many important applications. It also has generated algorithmic questions that have been in a state of extremely rapid development in the past 20 years. Altogether, the fastest algorithms that are now known for the problem are much faster, and some are much simpler, than the ones that were in use a short time ago, but it is still unclear how close to the 'ultimate' algorithm we are.

Definition. *A network is an edge-capacitated directed graph, with two distinguished vertices called the source and the sink.*

To repeat that, this time a little more slowly, suppose first that we are given a directed graph (*digraph*) *G*. That is, we are given a set of vertices, and a set of *ordered* pairs of these vertices, these pairs being the *edges* of the digraph. It is perfectly OK to have both an edge from *u* to *v* and an edge from *v* to *u*, or both, or neither, for all $u \neq v$. No edge (*u*, *u*) is permitted. If an edge *e* is directed *from* vertex *v to* vertex *w*, then *v* is the *initial* vertex of *e* and *w* is the *terminal* vertex of *e*. We may then write $v = Init(e)$ and $w = Term(e)$.

Next, in a network there is associated with each directed edge *e* of the digraph a positive real number called its *capacity*, and denoted by $cap(e)$. Finally, two of the vertices of the digraph are distinguished. One, *s*, is the source, and the other, *t*, is the sink of the network.

We will let **X** denote the resulting network. It consists of the digraph *G*, the given set of edge capacities, the source, and the sink. A network is shown in Fig. (1).

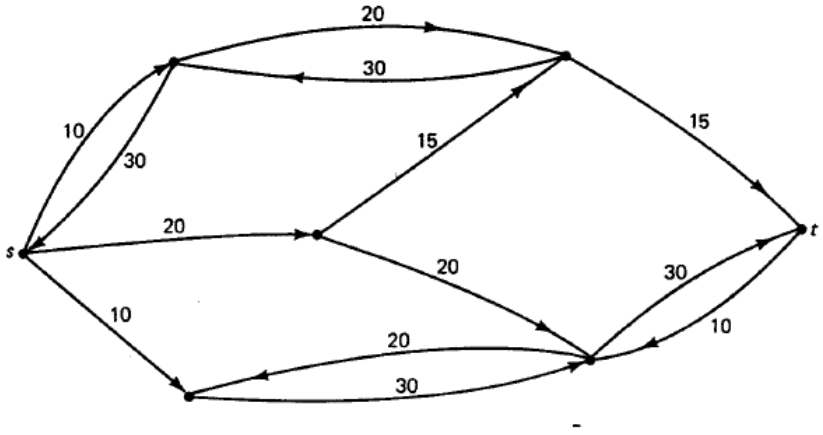


Figure (1)

Now roughly speaking, we can think of the edges of *G* as conduits for a fluid, the capacity of each edge being the carrying-capacity of the edge for that fluid. Imagine that the fluid flows in the network from the source to the sink, in such a way that the amount of fluid in each edge does not exceed the capacity of that edge. We want to know the maximum net quantity of fluid that could be flowing from source to sink. That was a rough description of the problem; here it is more precisely.

Definition. *A flow in a network* **X** *is a function f that assigns to each edge e of the network a real number*

f(*e*), *in such a way that*

(1) For each edge e we have $0 \leq f(e) \leq cap(e)$ *and*

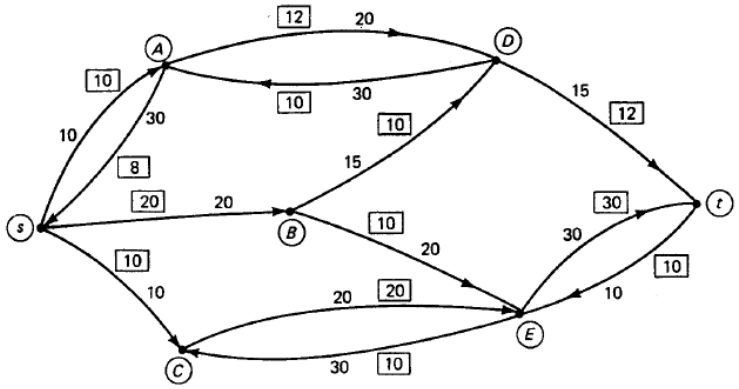
(2) For each vertex v other than the source and the sink, it is true that

$$\sum_{Init(e)=v} f(e) = \sum_{Term(e)=v} f(e). \qquad \ldots\ldots(1)$$

The condition (1) is a flow conservation condition. It states that the outflow from *v* (the left side of (1)) is equal to the inflow to *v* (the right side) for all vertices *v* other than *s* and *t*. In the theory of electrical networks such conservation conditions are known as Kirchhoff's laws. Flow cannot be manufactured anywhere in the network except at *s* or *t*. At other vertices, only redistribution or rerouting takes place.

Since the source and the sink are exempt from the conservation conditions there may, and usually will, be a nonzero net flow out of the source, and a nonzero net flow into the sink. Intuitively it must already be clear that these two are equal. If we let Q be the net outflow from the source, then Q is also the net inflow to the sink. The quantity Q is called the *value of the flow*.

In Fig. (2) there is shown a flow in the network of Fig. (1). The amounts of flow in each edge are shown in the square boxes. The other number on each edge is its capacity. The letter inside the small circle next to each vertex is the name of that vertex, for the purposes of the present discussion. The value of the flow in Fig. (2) is $Q = 32$.



Figure(2)

2. ***Algorithms for the network flow problem***

The first algorithm for the network flow problem was given by Ford and Fulkerson. They used that algorithm not only to solve instances of the problem, but also to prove theorems about network flow, a particularly happy combination. The speed of their algorithm, it turns out, depends on the edge capacities in the network as well as on the numbers V of vertices, and E of edges, of the network. Indeed, for certain (irrational) values of edge capacities they found that their algorithm might not converge at all.

In 1969 Edmonds and Karp gave the first algorithm for the problem whose speed is bounded by a polynomial function of E and V only. In fact that algorithm runs in time $O(E2V)$. Since then

there has been a steady procession of improvements in the algorithms, culminating, at the time of this writing anyway, with an $O(EV \log V)$ algorithm.

3. ***The algorithm of Ford and Fulkerson***

The basic idea of the Ford-Fulkerson algorithm for the network flow problem is this: start with some flow function (initially this might consist of zero flow on every edge). Then look for a *flow augmenting path* in the network. A flow augmenting path is a path from the source to the sink along which we can push some additional flow.

In Fig. (3) below we show a flow augmenting path for the network of Fig. (4). The capacities of the edges are shown on each edge, and the values of the flow function are shown in the boxes on the edges.

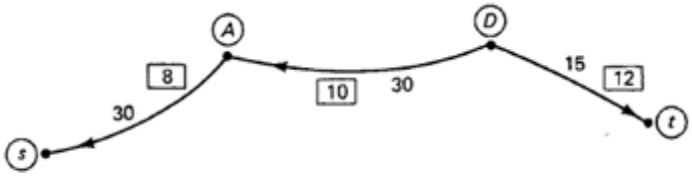
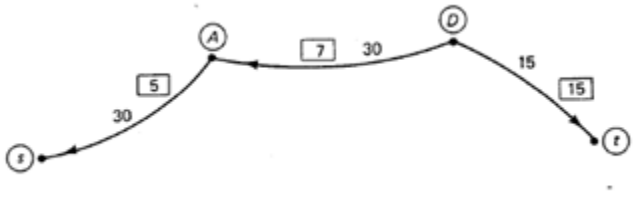


Figure (3)



Figure(4)

An edge can get elected to a flow augmenting path for two possible reasons. Either (a) the direction of the edge is *coherent* with the direction of the path from source to sink and the present value of the flow function on the edge is below the capacity of that edge, or (b) the

direction of the edge is *opposed* to that of the path from source to sink and the present value of the flow function on the edge is strictly positive.

Indeed, on all edges of a flow augmenting path that are coherently oriented with the path we can increase the flow along the edge, and on all edges that are incoherently oriented with the path we can decrease the flow on the edge, and in either case we will have *increased the value of the flow* (think about that one until it makes sense).

It is, of course, necessary to maintain the conservation of flow, *i.e.*, to respect Kirchhoff's laws. To do this we will augment the flow on every edge of an augmenting path by the same amount. If the conservation conditions were satisfied before the augmentation then they will still be satisfied after such an augmentation.

It may be helpful to remark that an edge is coherently or incoherently oriented only *with respect to a given path* from source to sink. That is, the coherence, or lack of it, is not only a property of the directed edge, but depends on how the edge sits inside a chosen path.

Thus, in Fig. (3) the first edge is directed towards the source, *i.e.*, incoherently with the path. Hence if we can *decrease* the flow in that edge we will have *increased* the value of the flow function, namely the net flow out of the source. That particular edge can indeed have its flow decreased, by at most 8 units. The next edge carries 10 units of flow towards the source. Therefore if we *decrease* the flow on that edge, by up to 10 units, we will also have *increased* the value of the flow function. Finally, the edge into the sink carries 12 units of flow and is oriented towards the sink. Hence if we *increase* the flow in this edge, by at most 3 units since its capacity is 15, we will have increased the value of the flow in the network.

Since every edge in the path that is shown in Fig. (3) can have its flow altered in one way or the other so as to increase the flow in the network, the path is indeed a flow augmenting path. The most that we might accomplish with this path would be to push 3 more units of flow through it from source to sink.

We couldn't push more than 3 units through because one of the edges (the edge into the sink) will tolerate an augmentation of only 3 flow units before reaching its capacity.

To augment the flow by 3 units we would decrease the flow by 3 units on each of the first two edges and increase it by 3 units on the last edge. The resulting flow in this path is shown in Fig. (4). The flow in the full network, after this augmentation, is shown in Fig. (5). Note carefully that if these augmentations are made then flow conservation at each vertex of the network will still hold.

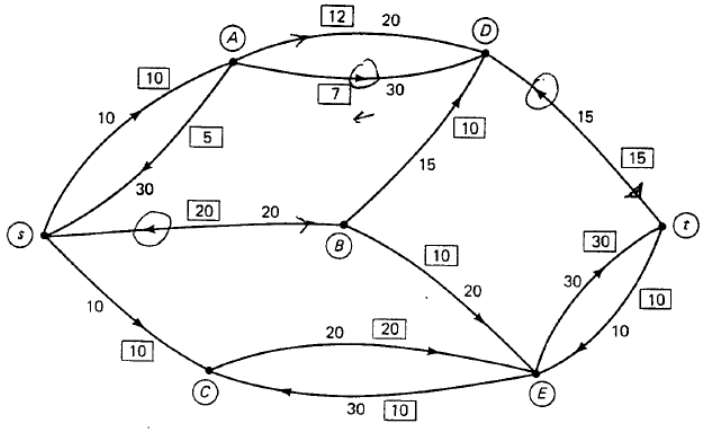


Figure (5)

After augmenting the flow by 3 units as we have just described, the resulting flow will be the one that is shown in Fig. (5). The value of the flow in Fig. (2) was 32 units. After the augmentation, the flow function in Fig. (5) has a value of 35 units.

We have just described the main idea of the Ford-Fulkerson algorithm. It first finds a flow augmenting path. Then it augments the flow along that path as much as it can. Then it finds another flow augmenting path, etc. The algorithm terminates when no flow augmenting paths exist. We will prove that when that happens, the flow will be at the maximum possible value, *i.e.*, we will have found the solution of the network flow problem. We will now describe the steps of the algorithm in more detail.

Definition. *Let f be a flow function in a network* **X**. *We say that an edge e of* **X** *is usable from v to w if either e is directed from v to w and the flow in e is less than the capacity of the edge, or e is directed from w to v and the flow in e is* > 0.

Now, given a network and a flow in that network, how do we find a flow augmenting path from the source to the sink? This is done by a process of labeling and scanning the vertices of the

network, beginning with the source and proceeding out to the sink. Initially all vertices are in the conditions 'unlabeled' and 'unscanned.' As the algorithm proceeds, various vertices will become labeled, and if a vertex is labeled, it may become *scanned*. To scan a vertex *v* means, roughly, that we stand at *v* and look around at all neighbors *w* of *v* that haven't yet been labeled. If *e* is some edge that joins *v* with a neighbor *w*, and if the edge *e* is usable from *v* to *w* as defined above, then we will label *w*, because any flow augmenting path that has already reached from the source to *v* can be extended another step, to *w*.

The label that every vertex *v* gets is a triple (*u*,±, *z*), and here is what the three items mean. The '*u*' part of the label of *v* is the name of the vertex that was being scanned when *v* was labeled. The '±' will be '+' if *v* was labeled because the edge (*u, v*) was usable from *u* to *v* (*i.e.*, if the flow from *u* to *v* was less than the capacity of (*u, v*)) and it will be '−' if *v* was labeled because the edge (*v, u*) was usable from *u* to *v* (*i.e.*, if the flow from *v* to *u* was > 0).

Finally, the '*z*' component of the label represents the largest amount of flow that can be pushed from the source to the present vertex *v* along any augmenting path that has so far been found. At each step the algorithm will replace the current value of *z* by the amount of new flow that could be pushed through to *z* along the edge that is now being examined, if that amount is smaller than *z*.

So much for the meanings of the various labels. As the algorithm proceeds, the labels that get attached to the different vertices form a record of how much flow can be pushed through the network from the source to the various vertices, and by exactly which routes.

To begin with, the algorithm labels the source with (*−1,+,1*). The source now has the label-status *labeled* and the scan-status *unscanned*. Next we will scan the source. Here is the procedure for scanning any vertex *u*.

procedure *scan*(*u*:vertex;**X** :network; *f*:flow);
for every 'unlabeled' vertex *v* that is connected
to *u* by an edge in either or both directions, do
if the flow in (*u, v*) is less than *cap*(*u, v*)
then

label *v* with (*u*,+, min{*z*(*u*), *cap*(*u*, *v*) − *flow*(*u*, *v*)})
else if the flow in (*v*, *u*) is > 0
then
label *v* with (*u*,−, min{*z*(*u*), *flow*(*v*, *u*)}) and
change the label-status of *v* to 'labeled';
change the scan-status of *u* to 'scanned'
end.*{scan}*

We can use the above procedure to describe the complete scanning and labeling of the vertices of the network, as follows.

procedure *labelandscan*(**X** :network; *f*:flow; *whyhalt*:reason);
give every vertex the scan-status '*unscanned*'
and the label-status '*unlabeled*';
u := *source*;
label *source* with (−*1*,+,*1*);
label-status of *source*:= '*labeled*';
while {there is a '*labeled*' and '*unscanned*' vertex *v*
and *sink* is '*unlabeled*'}
do *scan*(*v*,**X**, *f*);
if *sink* is *unlabeled*
then '*whyhalt*':='*flow is maximum*'
else '*whyhalt*':= '*it*'*s time to augment*'
end.*{labelandscan}*

Obviously the labeling and scanning process will halt for one of two reasons: either the sink *t* acquires a label, or the sink never gets labeled but no more labels can be given. In the first case we will see that a flow augmenting path from source to sink has been found, and in the second

case we will prove that the flow is at its maximum possible value, so the network flow problem has been solved.

Suppose the sink does get a label, for instance the label (*u*,±, *z*). Then we claim that the value of the flow in the network can be augmented by *z* units.

To prove this we will construct a flow augmenting path, using the labels on the vertices, and then we will change the flow by *z* units on every edge of that path in such a way as to increase the value of the flow function by *z* units. This is done as follows.

If the sign part of the label of *t* is '+,' then increase the flow function by *z* units on the edge (*u*, *t*), else decrease the flow on edge (*t*, *u*) by *z* units.

Then move back one step away from the sink, to vertex *u*, and look at its label, which might be (*w*,±, *z*1). If the sign is '+' then increase the flow on edge (*w*, *u*) by *z* units (not by *z*1 units!), while if the sign is '−' then decrease the flow on edge (*u*,*w*) by *z* units. Next replace *u* by *w*, etc., until the source *s* has been reached. A little more formally, the flow augmentation algorithm is the following.

procedure *augmentflow*(**X** :network; *f*:flow ; *amount*:real);
*{*assumes that *labelandscan* has just been done*}*
v:=*sink*;
amount:= the '*z*' part of the label of *sink*;
repeat
(*previous*, *sign*, *z*) :=*label*(*v*);
if *sign*='+'
then
increase *f*(*previous*, *v*) by *amount*
else
decrease *f*(*v*, *previous*) by *amount*;
v := *previous*
until *v*= *source*
end.*{augmentflow}*

The value of the flow in the network has now been *increased* by *z* units. The whole process of labeling and scanning is now repeated, to search for another flow augmenting path. The algorithm halts only when we are unable to label the sink. The complete Ford-Fulkerson algorithm is shown below.

procedure *fordfulkerson*(**X** :network; *f*: flow; *maxflowvalue*:real);
*{*finds maximum flow in a given network **X** *}*
set *f*:=0 on every edge of **X** ;
maxflowvalue:=0;
repeat
labelandscan(**X**, *f*, *whyhalt*);
if *whyhalt*=‘*it’s time to augment*’ **then**
augmentflow(**X**,*f*, *amount*);
maxflowvalue := *maxflowvalue* + *amount*
until *whyhalt* = ‘*flow is maximum*’
end.*{fordfulkerson}*

Let’s look at what happens if we apply the labelling and scanning algorithm to the network and flow shown in Fig. (2). First vertex *s* gets the label (*−1*,+,*1*). We then scan *s*. Vertex *A* gets the label (*s,−,* 8), *B* cannot be labeled, and *C* gets labeled with (*s*,+, 10), which completes the scan of *s*. Next we scan vertex *A*, during which *D* acquires the label (*A*,+, 8). Then *C* is scanned, which results in *E* getting the label (*C,−,* 10). Finally, the scan of *D* results in the label (*D*,+, 3) for the sink *t*.

From the label of *t* we see that there is a flow augmenting path in the network along which we can push 3 more units of flow from *s* to *t*. We find the path as in procedure *augment flow* above, following the labels backwards from *t* to *D*, *A* and *s*. The path in question will be seen to be exactly the one shown in Fig. (3), and further augmentation proceeds as we have discussed above.

Lecture (9)

APPROXIMATION ALGORITHMS

1. *Introduction*

There are many hard combinatorial optimization problems that cannot be solved *efficiently* using backtracking or randomization. An alternative in this case for tackling *some* of these problems is to devise an *approximation algorithm*, given that we will be content with a "reasonable" solution that approximates an optimal solution. Associated with each approximation algorithm, there is a performance bound that guarantees that the solution to a given instance will not be far away from the neighborhood of the exact solution. A marking characteristic of (most of) approximation algorithms is that they are fast, as they are mostly greedy heuristics. As stated in the proof of correctness of a greedy algorithm may be complex. In general, the better the performance bound the harder it becomes to prove the correctness of an approximation algorithm. This will be evident when we study some approximation algorithms. One should not be optimistic, however, about finding an efficient approximation algorithm, as there are hard problems for which even the existence of a "reasonable" approximation algorithm is unlikely unless NP = P.

The Euclidian Traveling Salesman Problem

In this section we consider the following problem. Given a set S of n points in the plane, find a tour τ on these points of shortest length. Here, a tour is a circular path that visits every point exactly once. This problem is a special case of the traveling salesman problem, and is commonly referred to as the EUCLIDEAN MINIMUM SPANNING TREE (EMST), which is known to be NP-hard.

Let p_1 be an arbitrary starting point. An intuitive method would proceed in a greedy manner, visiting first that point closest to p_1, say p_2, and then that point which is closest to p_2, and so on. This method is referred to as the *nearest neighbor* (NN) heuristic, and it can be shown that it does not result in a bounded performance ratio, i.e., $R_{NN} = \infty$. Indeed, it can be shown that this method results in the performance ratio

$$R_{NN}(I) = \frac{NN(I)}{OPT(I)} = O(\log n).$$

An alternative approximation algorithm can be summarized as follows.

First, a minimum cost spanning tree T is constructed.

Next, a multigraph T' is constructed from T by making two copies of each edge in T. Next, an Eulerian tour τ_e is found (an Eulerian tour is a cycle that visits every edge exactly once). Once τ_e is found, it can easily be converted into the desired Hamiltonian tour τ by tracing the Eulerian tour τ_e and deleting those vertices that have already been visited. Figure 1 illustrates the method. The input graph in Fig. 1 (a) is converted into an Eulerian multigraph in Fig. 1 (b). Figure 1 (c) shows the resulting tour after bypassing those points that have already been visited.

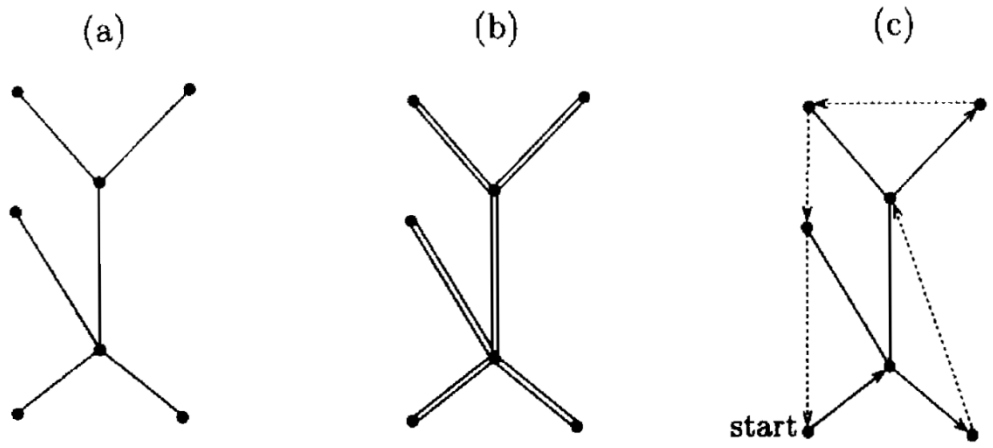


Fig. 1 An illustration of the approximation algorithm for the EUCLIDEAN MINIMUM SPANNING TREE.

Algorithm 1 ETSPAPPROX
Input: An instance I of EUCLIDEAN MINIMUM SPANNING TREE
Output: A tour τ for instance I.

1. Find a minimum spanning tree T of S.
2. Identify the set X of odd degree in T.
3. Find a minimum weight matching M on X.
4. Find an Eulerian tour τ_e in $T \cup M$.
5. Traverse τ_e edge by edge and bypass each previously visited vertex. Let τ be the resulting tour.

The time complexity of this algorithm is $O(n^3)$

The Vertex Cover Problem

Recall that a vertex cover C in a graph $G = (V, E)$ is a set of vertices such that each edge in E is incident to at least one vertex in C. We have shown in Sec. 10.4.2 that the problem of deciding whether a graph contains a vertex cover of size k, where k is a positive integer, is NP-complete. It follows that the problem of finding a vertex cover of minimum size is NP-hard.

Perhaps, the most intuitive heuristic that comes to mind is as follows. Repeat the following step until E becomes empty. Pick an edge e arbitrarily and add one of its endpoints, say v, to the vertex cover. Next, delete e and all other edges incident to v. Surely, this is an approximation algorithm that outputs a vertex cover. However, it can be shown that the performance ratio of this algorithm is unbounded. Surprisingly, if when considering an edge e, we add both of its endpoints to the vertex cover, then the performance ratio becomes 2. The process of picking an edge, adding its endpoints to the cover, and deleting all edges incident to these endpoints is equivalent to finding a *maximal* matching in G. Note that this matching need not be of maximum cardinality. This approximation algorithm is outlined in Algorithm VCOVERAPPROX.

Algorithm VCOVERAPPROX
Input: An undirected graph $G = (V, E)$.
Output: A vertex cover C for G.

1. $C \leftarrow \{\}$
2. **while** $E \neq \{\}$
3. Let $e = (u, v)$ be any edge in E.
4. $C \leftarrow C \cup \{u, v\}$
5. Remove e and all edges incident to u or v from E.
6. **end while**

The Knapsack Problem

Let $U = \{u_1, u_2, \ldots, u_n\}$ be a set of items to be packed in a knapsack of size C. For $1 \leq j \leq n$, let s_j and v_j be the size and value of the jth item, respectively. Recall that the objective is to fill the knapsack with some items in U whose total size is at most C and such that their total value is maximum (see Sec. 7.6). Assume without loss of generality that the size of each item is not larger than C.

Consider the greedy algorithm that first orders the items by decreasing value to size ratio (v_j/s_j), and then considers the items one by one for packing. If the current item fits in the available space, then it is included,

Algorithm KNAPSACKGREEDY
Input: $2n + 1$ positive integers corresponding to item sizes $\{s_1, s_2, \ldots, s_n\}$, item values $\{v_1, v_2, \ldots, v_n\}$ and the knapsack capacity C.
Output: A subset Z of the items whose total size is at most C.

1. Renumber the items so that $v_1/s_1 \geq v_2/s_2 \geq \ldots \geq v_n/s_n$.
2. $j \leftarrow 0$; $K \leftarrow 0$; $V \leftarrow 0$; $Z \leftarrow \{\}$
3. **while** $j < n$ and $K < C$
4. j $\leftarrow$ j+1
5. **if** $s_j \leq C - K$ **then**
6. $Z \leftarrow Z \cup \{u_j\}$
7. $K \leftarrow K + s_j$
8. $V \leftarrow V + v_j$
9. **end if**
10. **end while**
11. Let $Z' = \{u_s\}$, where u_s is an item of maximum size.
12. **if** $V \geq v_s$ **then return** Z
13. **else return** Z'.

Lecture (10)

SEARCHING

A fundamental operation intrinsic a great many computational tasks is *searching:* retrieving some particular information from a large amount of previously stored information. Normally we think of the information as divided up into records, each record haling a *key* for use in searching. The goal of the search is to find all records with keys matching a given *search key.* The purpose of the search is usually to access information within the record (not merely the key) for processing.

Two common terms often used to describe data structures for searching are *dictionaries* and *symbol tables.* For example, in an English language dictionary, the "keys" are the words and the "records" the entries associated with the words which contain the definition, pronunciation, and other associated information. (One can prepare for learning and appreciating searching methods by thinking about how one would implement a system allowing access to an English language dictionary.) A symbol table is the dictionary for a program: the "keys" a-e the symbolic names used in the program, and the "records" contain information describing the object named. In searching (as in sorting) we have: programs which are in widespread use on a very frequent basis, so that it will be worthwhile to study a variety of methods in some detail. As with sorting, we'll begin by looking at some elementary methods which are very useful for small tables and in other special situations and illustrate fundamental techniques exploited by more advanced methods. We'll look at methods which store records in arrays which are either searched with key comparisons or index: by key value, and we'll look at a fundamental method which builds structures defined by the key values.

As with priority queues, it is best to think of search algorithms as belonging to packages implementing a variety of generic operations which can be separated from particular implementations, so that alternate implementations could be substituted easily. The operations of interest include:

Initialize the data structure.

Search for a record (or records) having a given key.

Insert a new record.

Delete a specified record.

Join two dictionaries to make a large one.

Sort the dictionary; output all the records in sorted order.

As with priority queues, it is sometimes convenient to combine some of these operations. For example, a *search and insert* operation is often included for efficiency in situations where records with duplicate keys are not to be kept within the data structure. In many methods, once it has been determined that a key does not appear in the data structure, then the internal state of the search procedure contains precisely the information needed to insert a new record with the given key.

Records with duplicate keys can be handled in one of several ways, depending on the application. First, we could insist that the primary searching data structure contain only records with distinct keys. Then each "record" in this data structure might contain, for example, a link to a list of all records having that key. This is the most convenient arrangement from the point of view of the design of searching algorithms, and it is convenient in some applications since all records with a given search key are returned with one *search.* The second possibility is to leave records with equal keys in the primary searching data structure and return any record with the given key for a search. This is simpler for applications that process one record at a time, where the order in which records with duplicate keys are processed is not important. It is inconvenient from the algorithm design point of view because some mechanism for retrieving all records with a given key must still be provided. A third possibility is to assume that each record has a unique identifier (apart from the key), and require that *a search* find the record with a given identifier, given the key. Or, some more complicated mechanism could be used to distinguish among records with equal keys.

Each of the fundamental operations listed above has important applications, and quite a large number of basic organizations have been suggested to support efficient use of various combinations of the operations. In this and the next few chapters, we'll concentrate on implementations of the fundamental functions *search* and insert (and, of course, *initialize)*, with some comment on *delete* and *sort* when appropriate. As with priority queues, the *join* operation normally requires advanced techniques which we won't be able to consider here.

Sequential Searching

The simplest method for searching is simply to store the records in an array, then look through the array sequentially each time a record is sought. The following code shows an implementation of the basic functions using this simple organization.

```
type node=record key, info: integer end;
var a: array [0..maxN] of node;
    N: integer;
procedure initialize;
  begin N:=0 er d;
function seqsearch(v: integer; x: integer): integer;
  begin
  a[N+1].key:=v;
  if (x>=0) and (x<=N) then
    repeat x:=x+1 until v=a[x].key;
  seqsearch :=x
  end ;
function seqinsert(v: integer): integer;
  begin
  N:=N+1; a[N].key:=v;
  seqinsert:=N;
  end;
```

The code above processes records that have integer keys (key) and "associated information" (info). As with sorting, it will be necessary in many applications to extend the programs to handle more complicated records and keys, but this won't fundamentally change the algorithms. For example, info could be made into a pointer to an arbitrarily complicated record structure. In such a case, this field can serve as the unique identifier for the record for use in distinguishing among records with equal keys.

The search procedure takes two arguments in this implementation: the key value being sought and an index (x) into the array. The index is included to handle the case where several records have the same key value.

This method takes about N steps for an unsuccessful search (every record must be examined to decide that a record with any particular key is absent) and about N/2 steps, on the average, for a ***successful*** search (a "random" search for a record in the table will require examining about half the entries, on the average).

Binary Search

If the set of records is large, then the total search time can be significantly reduced by using a search procedure based on applying the "divide-and conquer" paradigm: divide the set of records into two parts, determine which of the two parts the key being sought belongs to, then concentrate on that part. A reasonable way to divide the sets of records into parts is to keep the records sorted, then use indices into the sorted array to delimit the part of the array being worked on. To find if a given key v is in the table, first compare it with the element at the middle position of the table. If v is smaller, then it must be in the first half of the table; if v is greater, then it must be in the second half of the table. Then apply the method recursively. (Since only one recursive call is involved, it is simpler to express the method iteratively.) This brings us directly to the following implementation, which assumes that the array a is sorted.

```
function binarysearch (v: integer) : integer;
  var x, l, r: integer;
  begin
  l:=1; r:=N;
  repeat
     x:=(l+r) div 2;
     if v<a[x].key then r:=x-1 else l:=x+1
  until (v=a[x].key) or (l>r);
  if v=a [x] .key then binarysearch :=x
                  else binarysearch := N+1
  end ;
```

Like Quick sort, this method uses the pointers 1 and r to delimit the sub file currently being worked on. Each time through the loop, the variable x is set to point to the midpoint of the current interval, and the loop terminates successfully, or the left pointer is changed to x+1, or the right pointer is changed to x-l, depending on whether the search value v is equal to, less than, or greater than the key value of the record stored at a[X]. The following table shows the sub files examined by this method when searching for (S) in a table built by inserting the keys

S E A R C H I N G E X A M P L E :

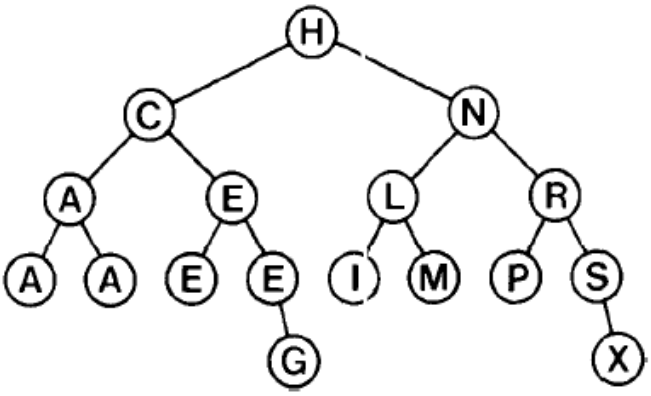
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	A	A	C	E	E	E	G	H	I	L	M	N	P	R	S	X
A	A	A	C	E	E	E	G	[H]	I	L	M	N	P	R	S	X
									I	L	M	[N]	P	R	S	X
													P	[R]	S	x
															[S]	x

The interval size is at least halved at each step, so the total number of times through the loop is only about 1og(N). However, the time required to insert new records is high: the array must be kept sorted, so records must be moved to make room for new records. For example, if the new record has a smaller key than any record in the table, then every entry must be moved over one position. A random insertion requires that N/2 records be moved, on the average. Thus, this method should not be used for applications which involve many insertions.

Some care must be exercised to properly handle records with equal keys for this algorithm: the index returned could fall in the middle of a block of records with key v, so loops which scan in both directions from that index should be used to pick up all the records. Of course, in this case the running time for the search is proportional to log N plus the number of records found.

The sequence of comparisons made by the binary search algorithm is predetermined: the specific sequence used is based on the value of the key being sought and the value of N. The

comparison structure can be simply described by a binary tree structure. The following binary tree describes the comparison structure for our example se, of keys:

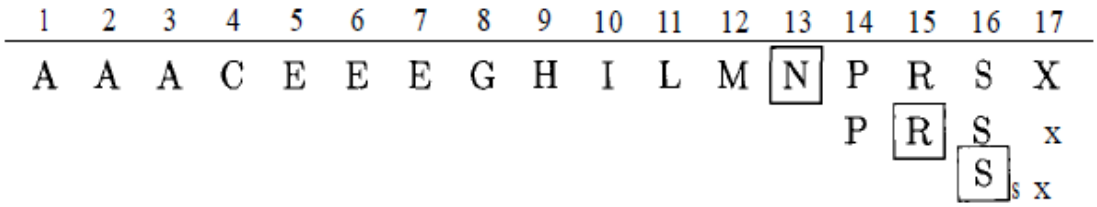


In searching for the key S for instance, it is first compared to H. Since it is greater, it is next compared to N; otherwise it would have been compared to C, etc. Below we will see algorithms that use an explicitly constructed binary tree structure to guide the search.

One improvement suggested for binary search is to try to guess more precisely where the key being sought falls within the current interval of interest (rather than blindly using the middle element at each step). This mimics the way one looks up a number in the telephone directory, for example: if the name sought begins with B, one looks near the beginning, but if it begins with Y, one looks near the end. This method, called *interpolation search,* requires only a simple modification to the program above. In the program above, the new place to search (the midpoint of the interval) is computed with the statement x:=(l+r) **div** 2. Interpolation search simply amounts to replacing i in this formula by an estimate of where the key might be based on the values available: i would be appropriate if v were in the middle of the interval between *a[I].key* and *a[r].key,* but we might have better luck trying

$$x := l + (v - a[l].key) * (r - l) \text{ div } (a[r].key - a[l].key).$$

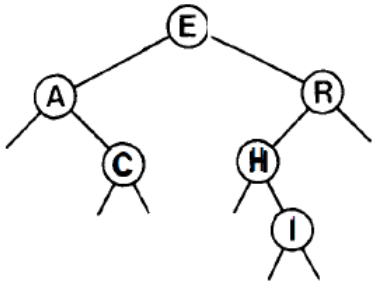
Of course, this assumes numerical key values. Suppose in our example that the ith letter in the alphabet is represented by the number i. Then, in a search for S, the first table position examined would be x = 1 + (19 - 1)*(17 - 1)/(24 - 1) = 14. The search is completed in just three steps:



Other search keys are found even more efficiently: for example X and A are found in the first step. Interpolation search manages to decrease the number of elements examined to about 1og logN. This is a very slowly growing function which can be thought of as a constant for practical purposes: if N is one billion, 1og log N < 5. Thus, any record can be found using only a few accesses, a substantial improvement over the conventional binary search method. But this assumes that the keys are rather well distributed over the interval, and it does require some computation: for small N, the 1og N cost of straight binary search is close enough to log log N that the cost of interpolating is not likely to be worthwhile. But interpolation search certainly should be considered for large files, for applications where comparisons are particularly expensive, or for external methods where very high access costs are involved.

Binary Tree Search

Binary tree search is a simple, efficient dynamic searching method which qualifies as one of the most fundamental algorithms in computer science. It's classified here as an "elementary" method because it is so simple; but in fact it is the method of choice in many situations. The idea is to build up an explicit structure consisting of nodes, each node consisting of a record containing a key and left and *right* links. The left and right links are either null, or they point to nodes called the left son and the *right* son. The sons are themselves the roots of trees, called the left *sub tree* and the *right* sub tree respectively. For example, consider the following diagram, where nodes are represented as encircled key values and the links by lines connected to nodes:



The links in this diagram all point down. Thus, for example, E's right link points to R, but H's left link is null. The defining property of a *tree* is that every node is pointed to by only one other node called its *father*. (We assume the existence of an imaginary node which points to the root.) The defining property of a binary *tree* is that each node has left and right links. For searching, each node also has a record with a key value; in a *binary* search *tree* we insist that all records with smaller keys are in the left sub tree and that all records in the right sub tree have larger (or equal) key values. We'll soon see that it is quite simple to ensure that binary search trees built by successively inserting new nodes satisfy this defining property.

A search procedure like binary search immediately suggests itself for this structure. To find a record with a give 1 key U, first compare it against the root. If it is smaller, go to the left sub tree; if it is equal, stop; and if it is greater, go to the right sub tree. Apply the method recursively. At each step, we're guaranteed that no parts of the tree other than the current sub tree could contain records with key v, and, just as the size of the interval in binary search shrinks, the "current sub tree" always gets smaller. The procedure stops either when a record with key v is found or, if there is no such record, when the "current sub tree" becomes empty. (The words "binary," "search," and "tree" are admittedly somewhat overuse at this point, and the reader should be sure to understand the difference between the binary search function given above and the binary search trees described here. Above, we used a binary tree to describe the sequence of comparisons made by a function searching in an array; here we actually construct 2 data structure of records connected with links which is used for the search.)

```
type link=↑node;
     node=record key, info: integer; l, r: link end;
var t, head, z: link;
function treesearch(v: integer; x: link): link;
  begin
  z↑.key:=v;
  repeat
    if v<x↑.key then x:=x↑.l else x:=x↑.r
  until v=x↑ . key;
  treesearch : =x
  end;
```

As with sequential list searching, the coding in this program is simplified by the use of a "tail" node z. Similarly, the insertion code given below is simplified by the use of a tree header node head whose right link points to the root. To search for a record with key v we set x:= tree search(v, head). If a node has no left (right) sub tree then its left (right) link is set to point to z. As in sequential search, we put the value sought in a two stop an unsuccessful search. Thus, the "current sub tree" pointed to by x never becomes empty and all searches are "successful" : the calling program can check whether the link returned points to a to determine whether the search was successful. It is sometimes convenient to think of links which point to z as pointing to imaginary *external* nodes with all unsuccessful searches ending at external nodes. The normal nodes which contain our keys are called *internal* nodes; by introducing external nodes we can say that every internal node points to two other nodes in the tree, even though, in our implementation, all of the external nodes are represented by the single node z.

For example, if D is sought in the tree above, first it is compared against E, the key at the root. Since D is less, it is next compared against A, the key in the left son of the node containing E. Continuing in this way; D is compared next against the C to the right of that node. The links in the node containing C are pointers to z so the search terminates with D being compared to itself in z and the search is unsuccessful.

Lecture (11)

STRING MATCHING ALGORITHM

Data to be processed often does not decompose logically into independent records with small identifiable pieces. This type of data is characterized only by the fact that it can be written down as a *string:* a linear (typically very long) sequence of characters.

Strings are obviously central in "word processing" systems, which provide a variety of capabilities for the manipulation of text. Such systems process *text strings,* which might be loosely defined as sequences of letters, numbers, and special characters. These objects can be quite large (for example, this book contains over a million characters), and efficient algorithms play an important role in manipulating them.

Another type of string is the *binary string, a* simple sequence of 0 and 1 values. This is in a sense merely a special type of text string, but it is worth making the distinction not only because different algorithms are appropriate but also binary strings arise naturally in many applications. For example, some computer graphics systems represent pictures as binary strings.

In one sense, text strings are quite different objects than binary strings, since they are made up of characters from a large alphabet. In another, the two types of strings are equivalent, since each text character can be represented by (say) eight binary bits and a binary string can be viewed as a text string by treating eight-bit chunks as characters. We'll see that the size of the alphabet from which the characters are taken to form a string is an important factor in the design of string processing algorithms.

A fundamental operation on strings is pattern matching: given a *text* string of length N and *a pattern* of length M, find an occurrence of the pattern within the text. (We will use the term "text" even when referring to a sequence of 0-l values or some other special type of string.) Most algorithms for this problem can easily be extended to find all occurrences of the pattern in the text, since they scan sequentially through the text and can be restarted at the point directly after the beginning of a match to find the next match.

The pattern-matching problem can be characterized as a searching problem with the pattern as the key, but the searching algorithms that we have studied do not apply directly because the pattern can be long and because it "lines up" with the text in an unknown way. It is an interesting problem to study because several very different (and surprising) algorithms have only recently been discovered which not only provide a spectrum of useful practical methods but also illustrate some fundamental algorithm design techniques.

Brute-Force Algorithm

The obvious method for pattern matching that immediately comes to mind is just to check, for each possible position in the text at which the pattern could match, whether it does in fact match. The following program searches in this way for the first occurrence of a pattern *p* [1. .M] in a text string a [1. .N] :

```
function brutesearch: integer;
  var i, j: integer;
  begin
  i:=1; j:=1;
  repeat
    if a[i]=p[j]
      then begin i:=i+1; j:=j+1 end
      else begin i:=i-j+2; j:=1 end;
  until (j>M) or (i>N);
  if j>M then brutesearch:=i-M else brutesearch:=i
  end;
```

The program keeps one pointer (i) into the text, and another pointer (j) into the pattern. As long as they point to matching characters, both pointers are incremented. If the end of the pattern is reached (j>M), then a match has been found. If *i* and j point to mismatching characters, then j is reset to point to the beginning of the pattern and *i* is reset to correspond to moving the pattern to the right one position for matching against the text. If the end of the text is reached *(i>N)*, then there is no match. If the pattern does not occur in the text, the value N+l is returned.

In a text-editing application, the inner loop of this program is seldom iterated, and the running time is very nearly proportional to the number of text characters examined. For example, suppose that we are looking for the pattern STING in the text string.

A STRING SEARCHING EXAMPLE CONSISTING OF SIMPLE TEXT

Then the statement j:=j+l is executed only four times (once for each S, but twice for the first ST) before the actual match is encountered. On the other hand, this program can be very slow for some patterns. For example, if the pattern is 00000001 and the text string is:

00000000000000000000000000000000000000000000000001

then j is incremented 7*45 (315) times before the match is encountered. Such degenerate strings are not likely in English (or Pascal) text, but the algorithm does run more slowly when used on binary (two-character) text, as might occur in picture processing and systems programming applications. The following table shows what happens when the algorithm is used to search for 10010111 in the following binary string:

```
10011101001001001001011100011 1
1001

    1
     10
       10010
          10010
             10010
                10010111
```

There is one line in this table for each time the body of the **repeat** loop is entered, and one character for each time j is incremented. These are the "false starts" that occur when trying to find the pattern: an obvious goal is to try to limit the number and length of these.

Knuth-Morris-Pratt Algorithm

The basic idea behind the algorithm discovered by Knuth, Morris, and Pratt is this: when a mismatch is detected, our "false start" consists of characters that we know in advance (since

they're in the pattern). Somehow we should be able to take advantage of this information instead of backing up the *i* pointer over all those known characters. For a simple example of this, suppose that the first character in the pattern doesn't appear again in the pattern (say the pattern is 10000000).

Then, suppose we have a false start j characters long at some position in the text. When the mismatch is detected, we know, by dint of the fact that j characters have matched, that we don't have to "back up" the text pointer i, since none of the previous j-l characters in the text can match the first character in the pattern. This change could be implemented by replacing i:=i-j+2 in the program above by i:=i+l. The practical effect of this change is limited because such a specialized pattern is not particularly likely to occur, but the idea is worth thinking about because the Knuth-Morris-Pratt algorithm is a generalization. Surprisingly, it is always possible to arrange things so that the i pointer is never decremented.

Fully skipping past the pattern on detecting a mismatch as described in the previous paragraph won't work when the pattern could match itself at the point of the mismatch. For example, when searching for 10100111 in 1010100111 we first detect the mismatch at the fifth character, but we had better back up to the third character to continue the search, since otherwise we would miss the match. But we can figure out ahead of time exactly what to do, because it depends only on the pattern, as shown by the following table:

j	$p[1..j-1]$	next $[j]$
2	1	1
3	10	1
4	101	2
5	1010	3
6	10100	1
7	101001	2
8	1010011	2

The array next [1..M] will be used to determine how far to back up when a mismatch is detected. In the table, imagine that we slide a copy of the first j-l characters of the pattern over

itself, from left to right starting with the first character of the copy over the second character of the pattern, stopping when all overlapping characters match (or there are none). These overlapping characters define the next possible place that the pattern could match, if a mismatch is detected at p[j]. The distance to back up (next [j]) is exactly one plus the number of the overlapping characters. Specifically, for j>l, the value of next[j] is the maximum k<j for which the first k-l characters of the pattern match the last k-l characters of the first j-l characters of the pattern. A vertical line is drawn just after p[j-next[j]] on each line of the table. As we'll soon see, it is convenient to define next[I] to be 0.

This next array immediately gives a way to limit (in fact, as we'll see, eliminate) the "backup" of the text pointer i: a generalization of the method above. When i and j point to mismatching characters (testing for a pattern match beginning at position i-j+1 in the text string), then the next possible position for a pattern match is beginning at position i-next[j]+l. But by definition of the next table, the first next[j]-1 characters at that position match the first next[j]-l characters of the pattern, so there's no need to back up the i pointer that far: we can simply leave the i pointer unchanged and set the j pointer to next [j], as in the following program:

```
function kmpsearch ; integer ;
  var i, j: integer;
  begin
  i:=1; j:=1;
  repeat
    if (j=0) or (a[i]=p[j])
      then begin i:=i+1; j:=j+1 end
      else begin j:=next[j] end;
  until (j>M) or (i>N);
  if j> M then kmpsearch : =i-M else kmpsearch : =i;
  end ;
```

When j=l and a[i] does not match the pattern, there is no overlap, so we want to increment i and set j to the beginning of the pattern. This is achieved by defining next [1] to be 0, which results in j being set to 0, then i is incremented and j set to 1 next time through the loop. (For this trick to work, the pattern array must be declared to start at 0.

Functionally, this program is the same as brutesearch, but it is likely to run faster for patterns which are highly self-repetitive. It remains to compute the next table. The program for this is short but tricky: it is basically the same program as above, except that it is used to match the pattern against itself.

```
procedure initnext ;
  var i, j: integer;
  begin
  i:=1; j:=0; next[1]:=0;
  repeat
    if (j=0) or (p[i]=p[j])
      then begin i:=i+1; j:=j+1; next[i]:=j end
      else begin j:=next[j] end;
  until i>M;
  end;
```

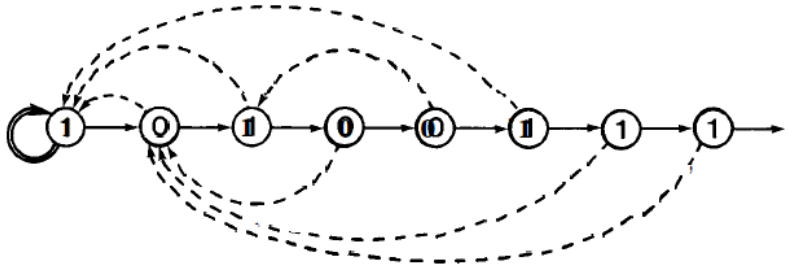
Just after i and j are incremented, it has been determined that the first j-l characters of the pattern match the characters in positions p [i-j- 1. .*i-1*], the last j-l characters in the first i-l characters of the pattern. And this is the largest j with this property, since otherwise a "possible match" of the pattern with itself would have been missed. Thus, j is exactly the value to be assigned to next [i].

An interesting way to view this algorithm is to consider the pattern as fixed, so that the next table can be "wired in" to the program. For example, the following program is exactly equivalent to the program above for the pattern that we've been considering, but it's likely to be much more efficient.

```
    i:=0;
  0: i:=i+1;
  1: if a[i]<>'1' then goto 0; i:=i+1;
  2: if a[i]<>'0' then goto 1; i:=i+1;
  3: if a[i]<>'1' then goto 1; i:=i+1;
  4: if a[i]<>'0' then goto 2; i:=i+1;
  5: if a[i]<>'0' then goto 3; i:=i+1;
  6: if a[i]<>'1' then goto 1; i:=i+1;
  7: if a[i]<>'1' then goto 2; i:=i+1;
  8: if a[i]<>'1' then goto 2; i:=i+1;
    search : =i-8;
```

The **goto** labels in this program correspond precisely to the next table. In fact, the in&next program above which computes the next table could easily be modified to output this program! To avoid checking whether i>N each time i is incremented, we assume that the pattern itself is stored at the end of the text as a sentinel, in a[N+l ..N+M]. (This optimization could also be applied to the standard implementation.) This is a simple example of a "string-searching compiler" : given a pattern, we can produce a very efficient program which can scan for that pattern in an arbitrarily long text string.

The program above uses just a few very basic operations to solve the string searching problem. This means that it can easily be described in terms of a very simple machine model, called a finite-state machine. The following diagram shows the finite-state machine for the program above:



The machine consists of *states* (indicated by circled letters) and *transitions* (indicated by arrows). Each state has two transitions leaving it: a match transition (solid line) and a non-match transition (dotted line). The states are where the machine executes instructions; the transitions are the goto instructions. When in the state labeled "5," the machine can perform just one instruction: "if the current character is x then scan past it and take the match transition, otherwise take the non-match transition." To "scan past" a character means to take the next character in the string as the "current character"; the machine scans past characters as it matches them. There is one exception to this: the non-match transition in the first state (marked with a double line) also requires that the machine scan to the next character.

(Essentially this corresponds to scanning for the first occurrence of the first character in the pattern.) In the next chapter we'll see how to use a similar (but more powerful) machine to help develop a much more powerful pattern-matching algorithm. The alert reader may have noticed that there's still some room for improvement in this algorithm, because it doesn't take

into account the character which caused the mismatch. For example, suppose that we encounter 1011 when searching for our sample pattern 10100111. After matching 101, we find a mismatch on the fourth character, at which point the *next* table says to check the second character, since we already matched the 1 in the third character. However, we could not have a match here: from the mismatch, we know that the next character in the text is not 0, as required by the pattern.

Another way to see this is to look at the version of the program with the next table "wired in": at label 4 we go to 2 if ***a[i]*** is not 0, but at label 2 we go to 1 if a[i] is not 0. Why not just go to 1 directly? Fortunately, it is easy to put this change into the algorithm. We need only replace the statement next[i] :=j in the initnext program by

```
if p[j]<>p[i] then next[i]:=j else next[i]:=next[j];
```

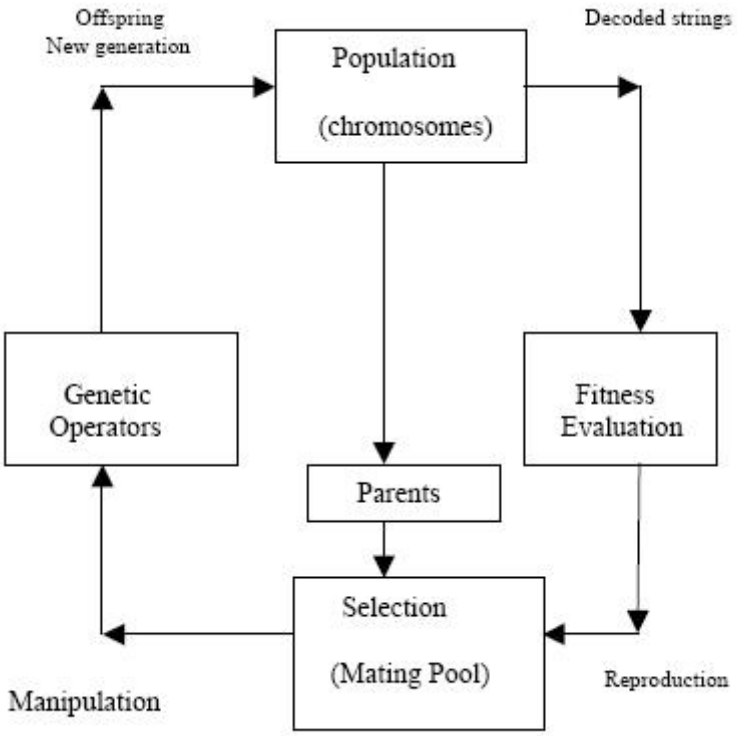
With this change, we either increment j cr reset it from the ***next*** table at most once for each value of i, so the algorithm is clearly linear.

The Knuth-Morris-Pratt algorithm LS not likely to be significantly faster than the brute-force method in most actual applications, because few applications involve searching for highly self-repetitive patterns in highly self repetitive text. However, the method does have a major virtue from a practical point of view: it proceeds sequentially through the input and never "backs up" in the input. This makes the method convenient for use on a large file being read in from some external device. (Algorithms which require backup require some complicated buffering in this situation.)

Lecture (12)

GENETIC ALGORITHM

The basic algorithm by which GAs operate is reasonably well established. What is important is that the general algorithm is followed and the evolutionary techniques that underlie it are understood. The cycle of a Genetic Algorithms is presented below:



The following sections discuss the need for each of these steps in terms of their relevance to evolutionary processes.

Population

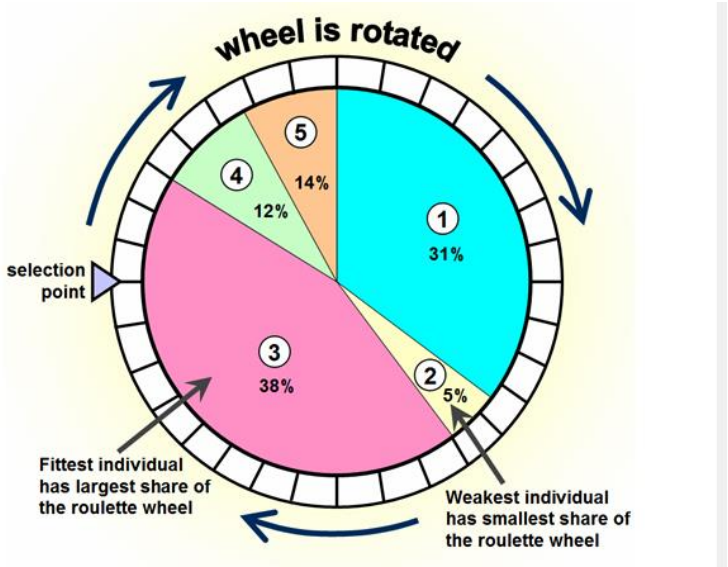
A population is initiated of legal solutions, selected by choosing random input values. There are no fixed rules for how large the population should be. The answer is dependent upon the type of problem. For a simple problem with a regular search space a small population of 40 to 100 will probably be sufficient. For larger more

complex problems and especially those with an irregular search space larger populations of 400 or more are recommended.

Fitness

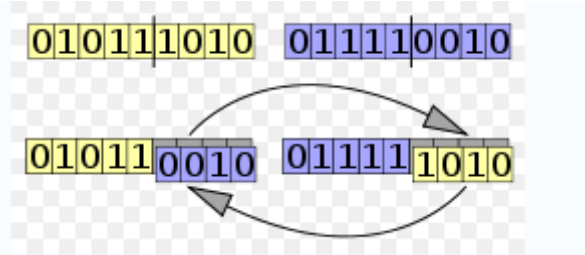
The fitness of individual chromosomes is a relative matter. For example when maximising a function; if one individual has a higher value, once processed by the function, than another then that individual is considered fitter. Things get a little more involved with multi-criteria problems. In these cases comparisons can be carried out to see if an individual dominates other members of a population by taking all criteria into consideration. If they do they are considered fitter. The most dominant, i.e. those who dominate all others, are referred to as a solutions. These are considered as candidate solutions to whatever problem you are trying to solve.

Selection of the Fittest



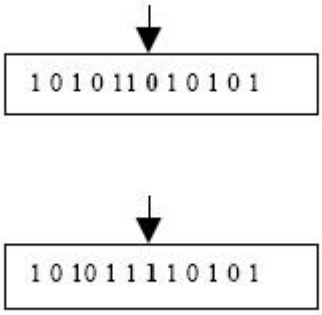
GAs operate over a number of generations. Following the evolutionary theme of this method, this means fitter solutions will tend to survive to the next generation. The selection method employed by many approaches is the **roulette wheel** selection process. In nature all individuals have a chance of surviving from one generation to the next – fitter solutions (i.e. those most dominant) have a better chance. Weaker more dominated individuals have a smaller chance (still an opportunity) of surviving.

Crossover



Nature generates the next generation using a mating process. As a result two parents create offspring, who consist of the genetic material of both parents. These offspring can be weaker or fitter than their parents (or similar). If they are weaker they will tend to die out – if they are stronger their chances of survival are better. GAs try to replicate this using a **crossover** operator. This emulates the mating process by exchanging chromosome patterns between individuals to create offspring for the next generation.

Mutation



Mutation exists in nature and causes an *unanticipated* change in a chromosome pattern. This can result in a much weakened individual and occasionally a much stronger one. Either way the principle behind mutation from an evolutionary point of view is that it occurs rarely, spontaneously and without reference to any other individual in the population. If the change is beneficial to the general population then that individual will tend to survive and will pass this trait on in future replication processes. Because of the way that GAs represent individuals this process is a very simple one and a typical **mutation** operator is relatively easy to

implement. It is important to remember that these processes occur very infrequently otherwise they would have a disruptive effect on the overall population.

There are no definitive methods of establishing how many generations a GA should run for. Simple problems may converge on good solutions after only 20 or 30 generations. More complex problems may need more. It is not unusual to run a GA for 400 generations for more complex problems such as job shops. The above figure suggests 100 generations. The most reliable method of deciding on this is trial and error, although a number of authors have suggested methods for determining how long a solution should live. The following algorithm is list the whole steps of GA.

```
Choose an initial population of chromosomes;
while termination condition not satisfied do
    repeat
        if crossover condition satisfied then
        {select parent chromosomes;
        choose crossover parameters;
        perform crossover};
        if mutation condition satisfied then
        {select chromosome(s) for mutation;
        {choose mutation points;
        perform mutation};
        evaluate fitness of offspring
    until sufficient offspring created;
    select new population;
endwhile
```