# 2-D Convex Hull

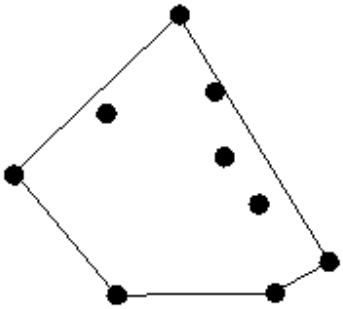
**Prerequisite**

- Geometry

**The Abstraction**

Given: A collection of points in the plane, find the convex polygon with smallest area such that each point is contained within (or on the boundary of) the polygon.

Observe that the vertices of such a polygon will be points from the given collection.



**Sample Problem: Cow Herding**

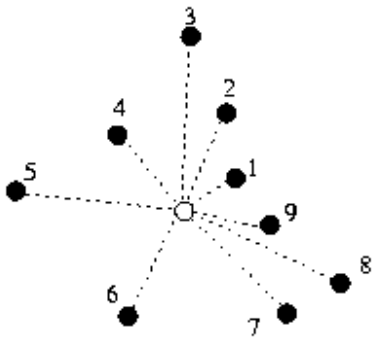
Farmer John wants a fence to keep the pesky local college students from tipping his cows over as they sleep. Each of his cows has a favorite spot for grazing grass, and Farmer John would like the fenced enclosure to include all of these favorite locations. Farmer John would like to enclose a convex figure, as it makes it a lot easier for the cows to make it to their grazing locations.

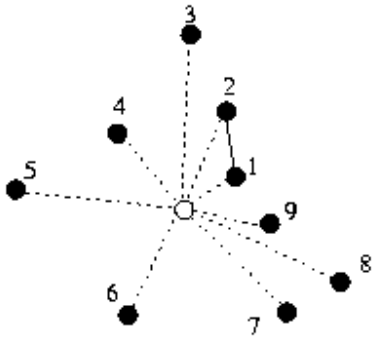
Help Farmer John by calculating the fence which encloses the minimum amount of area while still including all of the cows' favorite dining locations.

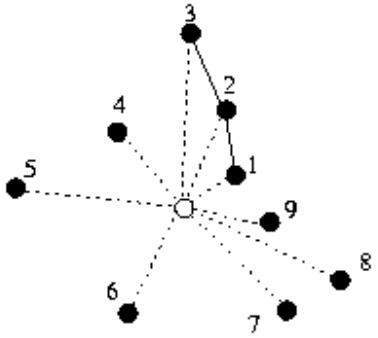
**The Algorithm**

Several algorithms solve the two dimensional convex hull algorithm. Here, we'll present only the ``gift-wrapping'' algorithm, which is probably the easiest to code and the easiest to remember.

The basic idea is to add the points in clockwise or counterclockwise order around some point within the final answer, checking to see if any angles greater than 180 degrees are created, which would make the final figure concave. Every time three points in a row appear such that the angle created by those three points is greater than 180 degrees, delete the middle point. Check the angle is done by calculating the cross product of vectors along two consecutive edges.




Find a point which will be within the convex hull:

- Calculate the angle that each point makes with the x axis (within the range 0 to 360 degrees)
- Sort the points based on this angle
- Add the first two points
- For every other point except the last point
- Make it the next point in the convex hull
- Check to see if the angle it forms with the previous two points is greater than 180 degrees
    - As long as the angle formed with the last two points is greater than 180 degrees, remove the previous point
- To add the last point
    - Perform the deletion above,
    - Check to see if the angle the last point forms with the previous point and the first point is greater than 180 degrees or if the angle formed with the last point and the first two points is greater than 180 degrees.

    - If the first case is true, remove the last point, and continue checking with the next-to-last point.
    - If the second case is true, remove the first point and continue checking.
    - Stop when neither case is true.
- The adding of the points is linear time, as is the calculation of the angles. Thus, the run-time is dominated by the time of the sort, so gift-wrapping runs in O($n$ log $n$) time, which is provably optimal.

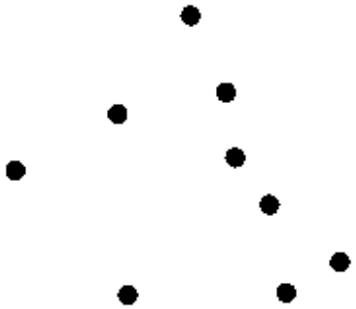
## Pseudocode

Here is the pseudocode for this convex hull algorithm:

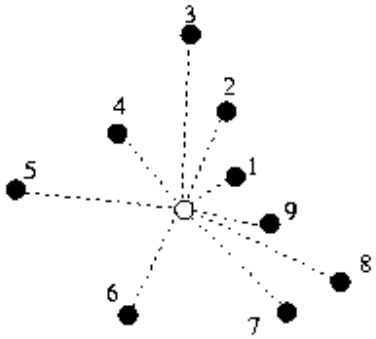
```
 # x(i), y(i) is the x,y position
     #    of the i-th point
     # zcrossprod(v1,v2) -> z component
     #             of the vectors v1, v2
     #  if zcrossprod(v1,v2) < 0,
     #     then v2 is "right" of v1
     #   since we add counter-clockwise
     #     <0 ->  angle > 180 deg
 1   (midx, midy) = (0, 0)
 2   For all points i
 3     (midx, midy) = (midx, midy) +
           (x(i)/npoints, y(i)/npoints)
 4   For all points i
 5     angle(i) = atan2(y(i) - midy,
                        x(i) - midx)
 6     perm(i) = i

 7   sort perm based on the angle() values
     # i.e., angle(perm(0)) <=
              angle(perm(i)) for all i

     # start making hull
 8   hull(0) = perm(0)
 9   hull(1) = perm(1)
10   hullpos = 2
11   for all points p, perm() order,
                except perm(npoints - 1)
12     while (hullpos > 1 and
           zcrossprod(hull(hullpos-2) -
13         hull(hullpos-1),
           hull(hullpos-1) - p) < 0)
14           hullpos = hullpos - 1
15       hull(hullpos) = p
16       hullpos = hullpos + 1

     # add last point
17   p = perm(npoints - 1)
18   while (hullpos > 1 and
          zcrossprod(hull(hullpos-2) -
19        hull(hullpos-1),
          hull(hullpos-1) - p) < 0)
20     hullpos = hullpos - 1

21   hullstart = 0
22   do
23     flag = false
24     if (hullpos - hullstart >= 2 and
            zcrossprod(p -
25          hull(hullpos-1),
            hull(hullstart) - p) < 0)
26       p = hull(hullpos-1)
27       hullpos = hullpos - 1
28       flag = true
29     if (hullpos - hullstart >= 2 and
           zcrossprod(hull(hullstart) - p,
           hull(hullstart+1) -
           hull(hullstart)) < 0)
30       hullstart = hullstart + 1
31       flag = true
32   while flag
33   hull(hullpos) = p
34   hullpos = hullpos + 1
```

## Sample Run

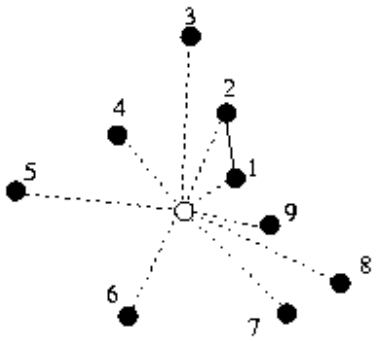
**For the sample run, use these points:**



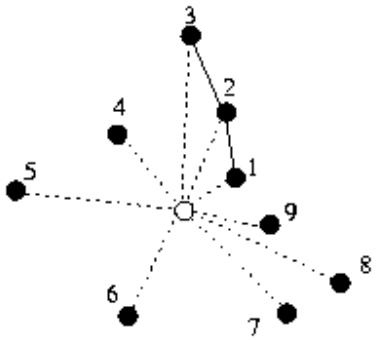
**Select a center, calculate angles, and sort.**



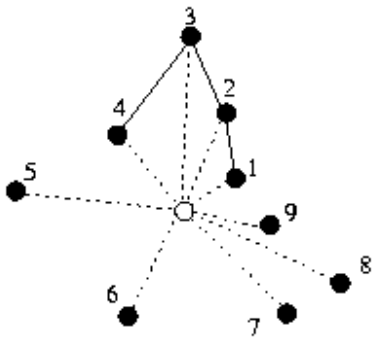

**Now, start by adding the first two points.**



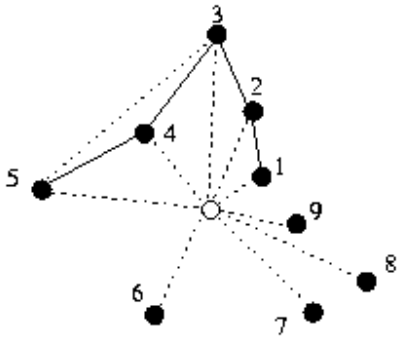

**Now, add the third point. Since this does not create an angle of greater than 180 degrees with the first two points, we just have to add the point.**



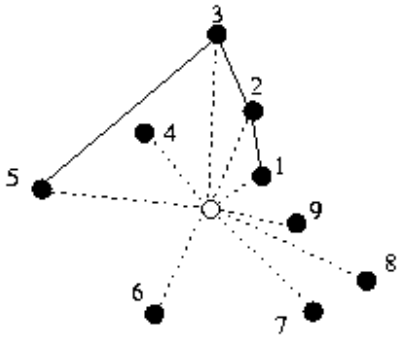

**Add the fourth point. Again, no angle greater than 180 degrees was created, so no further work is necessary.**




**Add the fifth point.**

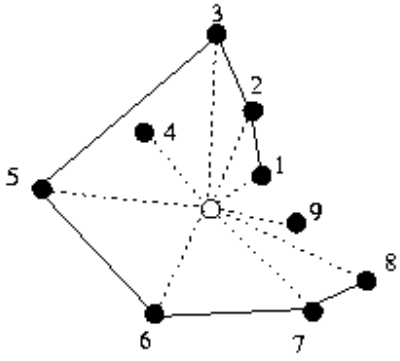



**Since the third, fourth, and fifth points together create an angle of greater than 180 degrees (a "right" turn), we remove the fourth point.**

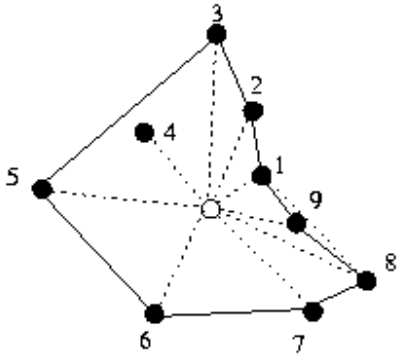



**The second, third, and fifth points do not create an angle of greater than 180 degrees, so we are done with adding the fifth point.**

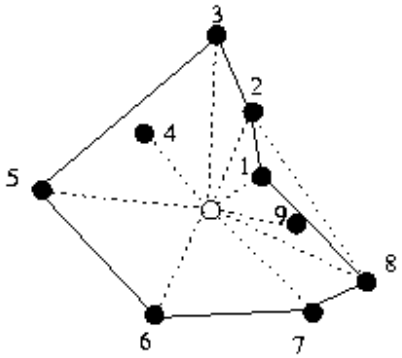
**Add the sixth, seventh, and eighth points. None of these require additional work.**



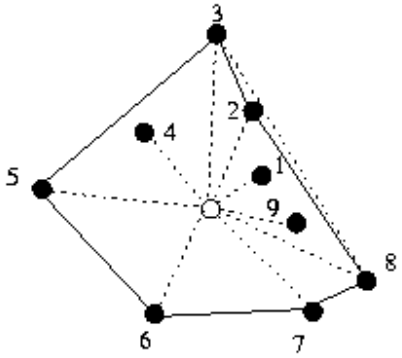
**Next, add the last point.**



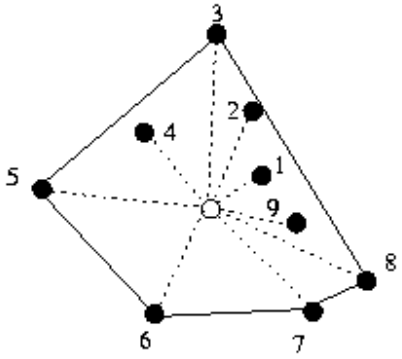

**The eighth, ninth, and first point create a "right" turn; remove the ninth point.**




**The seventh, eighth, and first points are fine, but the eighth, first and second points have a "right" term, so we must remove the first point.**




**Now the eighth, second and thirst points have a "right" term, so we must remove the second point.**




**No more violations exist, so we are done, and we have the convex hull of the given points.**

### Problem Cues

Problems which ask for enclosing points within a polygon are usually convex hull problems. If the problem asks for a minimum area convex polygon or a polygon of minimum circumference, it's almost certainly asking for the convex hull.

### Extensions

Unfortunately, this algorithm does not extend in an obvious manner to three dimensions. Fortunately, the three dimensional algorithms are all fairly complicated (four and higher+ dimensions are just plain ugly), so it's unlikely you'll get asked to do it there.

This algorithm no longer works if you limit the created polygon in any way (e.g., no more than n points or must be a rectangle).

### Sample Problems

#### Trees Problem [IOI 1991, problem 2]

Given: a collection of trees, surround it with wire such that you use the minimal amount of wire. Calculate the trees that will be the vertices of the polygon, the length of wire required, and whether the farmer's house, which is at a given location, lies inside of, outside of, or across the polygon.

Analysis: The vertices of the polygon and the length of wire required follow fairly directly above. The farmer's house is specified as an axis-aligned rectangle, so it takes a bit of geometry to determine if all the points are within the convex hull, without the convex hull, or some are within and some without, which gives you the answer you wanted. See the Geometry pamphlet for clues on these kinds of intersections.

### Cheapskate Moat Building

Given: A collection of polygon houses, calculate the minimum length moat that encloses all but at most one of them.

Analysis: To enclose a given polygon in convex polygon is equivalent to enclosing all of the vertices of the given polygon. This is very slightly a combination problem, which requires both a loop and a convex hull builder. For each house, delete the house and enclose the remaining vertices in a convex hull. Pick the house whose resulting convex hull is smallest. Note that only deleting houses which share a vertex with the convex hull of the entire set would help if some of the houses don't share a vertex with that convex hull.

USACO Gateway | Comment or Question

# Big Numbers

### Sample Problem: Factorial

Given N (1 <= N <= 200), calculate N! exactly.

### The Problem

All built-in integers have a limited value range. Sometimes, a problem will ask for the calculation of a number which is outside that range for all the available numbers. For example, 200! has 375 digits and would require a 1246 bit number (if one chose to store the number in binary) to hold it, which is unlikely to be available as a native datatype in contest environments. Thus, what is needed is a way to store and manipulate large numbers.

### The Structure

One method to store these numbers is actually fairly straight-forward: a list of numbers and a sign. This list can be either an array, if an upper bound is known on the length of the numbers, or a linked list, if the numbers have no upper bound.

One way to think of this storage method is keeping the `digits' of a number in a huge base.

Let $b$ be the base in which the number is stored, and $a_0$, $a_1$, ..., an be the digits stored. Then, the number represented is $(-1)^{sign}$ *times* $(a_0 + b^1 a_1 + b^2 a_2 + \ldots + b^n a_n)$. Note that $a_0, a_1, \ldots, a_n$ must be in the range 0..$b$-1.

Generally, the base $b$ is selected to be a power of 10, as it makes displaying the number very easy (but don't forget the leading zeroes).

Note that this representation stores the numbers in the order presented: $a_0$, then $a_1$, then $a_2$, etc. This is the reverse of the obvious ordering, and, for linked lists, it may be worthwhile to keep a deque, as some of the algorithms will want to walk through the list in the opposite order.

### Operations

For this data structure, figuring out how to do the various operations requires recalling how to do the operations by hand. The main problem that one has to be aware of is overflow. Always make sure that every addition and multiplication will not result in an overflow, or the entire operation will be incorrect.

For simplicity, the algorithms presented here will assume arrays, so if a number is $a_0$, $a_1$, ..., $a_k$, then for all $i > k$, $a_i = 0$. For linked lists, the algorithms become a bit more

difficult. In addition, the result bignums are assumed to be initialized to be 0, which will not be true in most cases.

#### Comparison

To compare two numbers $a_0$, $a_1$, ..., $a_n$ and $b_0$, $b_1$, ..., $b_k$, with signs signA and signB goes as follows:

```
# note that sign
# sizeA = number of digits of A
# signA = sign of A
# (0 => positive, 1 => negative)
 1   if (signA < signB)
 2     return A is smaller
 3   else if (signA > signB)
 4     return A is larger
 5   else
 6     for i = max(sizeA,sizeB) to 0
 7       if (a(i) > b(i))
 8         if (signA = 0)
 9           return A is larger
10        else
11           return A is smaller
12     return A = B
```

### Addition

Given two numbers $a_0$, $a_1$, ..., $a_n$ and $b_0$, $b_1$, ..., $b_k$, calculate the sum and store it in $c$. In order for addition to be possible, $2 \times b$ must be smaller than the largest representable number.

If the numbers have opposite signs, then: calculate which is larger in absolute value, subtract the smaller from the larger, and set the sign to be the same as the larger number. Otherwise, simply add the numbers from 0 to $max ( n, k)$, maintaining a carry bit.

Note that if it is known that both numbers have positive sign, the operation becomes much simpler and doesn't even require the writing of `absolute_subtract`.

Here is the pseudocode for addition:

```
 1 absolute_subtract(bignum A, bignum B,
                           bignum C)
 2   borrow = 0
 3   for pos = 0 to max(sizeA, sizeB)
 4     C(pos) = A(pos)-B(pos)-borrow
 5     if (C(pos) < 0)
 6       C(pos) = C(pos) + base
 7       borrow = 1
 8     else
 9       borrow = 0
    # it has to be done this way,
    # to handle the case of
    # subtracting two very close nums
    # (e.g., 7658493 - 7658492)
10     if C(pos) != 0
11       sizeC = pos
12   assert (borrow == 0,
       "|B| > |A|; can't handle this")

13 absolute_add(bignum A,
                  bignum B, bignum C)
14   carry = 0
15   for pos = 0 to max(sizeA,sizeB)
16     C(pos) = A(pos)+B(pos)+carry
17     carry = C(pos) / base
18     C(pos) = C(pos) mod base
19   if carry != 0
20     CHECK FOR OVERFLOW
21     C(max(sizeA,sizeB)+1) = carry
```

```
22     sizeC = max(sizeA,sizeB)+1
23   else
24     sizeC = max(sizeA, sizeB)

25 add (bignum A, bignum B, bignum C)
26   if signA == signB
27     absolution_add(A,B)
28     signC = signA
29   else
30     if (Compare(A,B) = A is larger)
       then
31       absolute_subtract(A,B)
32       signC = signA
33     else
34       absolute_subtract(B,A)
35       signC = signB
```

**Subtraction**

Subtraction is simple, given the addition operation above. To calculate A - B, negate the sign of B and add A and (-B).

**Multiplication by scalar**

To multiply a bignum A by the scalar s, if $|s \times b|$ is less than the maximum number representable, can be done in a similar manner to how it is done on paper.

```
 1   if (s < 0)
 2     signB = 1 - signA
 3     s = -s
 4   else
 5     signB = signA
 6   carry = 0
 7   for pos = 0 to sizeA
 8     B(pos) = A(pos) * s + carry
 9     carry = B(pos) / base
10     B(pos) = B(pos) mod base
11   pos = sizeA+1
12   while carry != 0
13     CHECK OVERFLOW
14     B(pos) = carry mod base
15     carry = carry / base
16     pos = pos + 1
17    sizeB = pos - 1
```

**Multiplication of two bignums**

Multiplying two numbers, if $b^2$ is below the largest representable number is a combination of scalar multiplication and in-place addition.

Basically, multiply one of the numbers by each digit of the other, and add it (with the appropriate offset) to a running total, the exact same way one does long multiplication on paper.

```
 1 multiply_and_add(bignum A, int s,
                int offset, bignum C)
 2   carry = 0
 3   for pos = 0 to sizeA
 4     C(pos+offset) = C(pos+offset) +
                   A(pos) * s + carry
 5     carry = C(pos+offset) / base
 6     C(pos+offset) =
              C(pos+offset) mod base
 7   pos = sizeA + offset + 1
 8   while carry != 0
 9     CHECK OVERFLOW
10     C(pos) = C(pos) + carry
11     carry = C(pos) / base
12     C(pos) = C(pos) mod base
13     pos = pos + 1
14   if (sizeC < pos - 1)
15     sizeC = pos - 1
```

```
16 multiply(bignum A,
           bignum B, bignum C)
17   for pos = 0 to sizeB
18     multiply_and_add(A,
                    B(pos), pos, C)
19   signC = (signA + signB) mod 2
```

**Division by scalar**

In order to divide the bignum *b* by a scalar *s*, *s x b* must be representable. Division is done in a very similar manner to long division.

```
20 divide_by_scalar (bignum A,
                int s, bignum C)
21   rem = 0
22   sizeC = 0
23   for pos = sizeA to 0
24     rem = (rem * b) + A(pos)
25     C(pos) = rem / s
26     if C(pos) > 0 and
              pos > sizeC then
27       sizeC = pos
28     rem = rem mod s
# remainder has the remainder
# of the division
```

**Division by bignum**

This is similar to division by scalar, except the division is done by multiple subtractions. Note that if *b* is large, this particular formulation takes too long.

```
 1 divide_by_bignum (bignum A,
             bignum B, bignum C)
 2   bignum rem = 0
 3   for pos = sizeA to 0
 4     mult_by_scalar_in_place(rem, b)
 5     add_scalar_in_place(rem,
                              A(pos))
 6     C(pos) = 0
 7     while (greater(rem, B))
 8       C(pos) = C(pos) + 1
 9       subtract_in_place(rem, B)
10     if C(pos) > 0 and pos > sizeC
       then
11       sizeC = pos
```

**Binary Search**

Binary search is a very helpful thing in general, but in particular for bignums, as operations are expensive. Given an upper and lower bound, check the mean of those two to see if it is above or below those bounds, and cut the range by (almost) a factor of 2.

For example, if *b* is large, then division by a bignum is slow to do by the method above, but the following works well:

```
 1 divide_by_bignum2 (bignum A,
                bignum B, bignum C)
 2   bignum rem = 0
 3   for pos = sizeA to 0
 4     mult_by_scalar_in_place(rem, b)
 5     add_scalar_in_place(rem,
                           A(pos))

 6     lower = 0
 7     upper = s-1
 8     while upper > lower
 9       mid = (upper + lower)/2 + 1
10       mult_by_scalar(B, mid, D)
```

```
11       subtract(rem, D, E)
12       if signE = 0
13         lower = mid
14       else
15         upper = mid - 1

16     C(pos) = lower
17     mult_by_scalar(B, lower, D)
18     subtract_in_place(rem, D)

19     if C(pos) > 0 and
                     pos > sizeC
20       sizeC = pos
```

USACO Gateway | Comment or Question

# Binary Numbers

### Representing Binary Numbers

Computers operate on 1's and 0's; these are called 'bits'. A byte is a group of 8 bits, like this example: 00110101. A computer word on a 32-bit computer ('int') is 4 bytes, 32 bits: 10011010110101011010001010101011. Other computers have different word sizes; over time, 64-bit integers will become more common.

As you can see, 32 ones and zeroes is a bit cumbersome to write down (or even to read). Thus, people conventionally break these large numbers of digits down into groups of 3 or 4 bits:

```
  1001.1010.1101.0101.1010.0010.1010.1011    <-- four bit groups
  10.011.010.110.101.011.010.001.010.101.011 <-- three bit groups
                           (note that the count of 3 starts on the right)
```

These grouped sets of bits are then mapped onto digits, either four bits per hexadecimal (base 16) digit or three bits per octal (base 8) digit. Obviously, hexadecimal needs some new digits (since decimal digits only go 0..9 and 6 more are needed). These days, the letters 'A'..'F' are used for the 'digits' that represent 10..15. Here's the map; the correspondence is obvious:

```
    OCTAL:                    HEXADECIMAL:
  000 -> 0  100 -> 4       0000 -> 0  0100 -> 4  1000 -> 8  1100 -> C
  001 -> 1  101 -> 5       0001 -> 1  0101 -> 5  1001 -> 9  1101 -> D
  010 -> 2  110 -> 6       0010 -> 2  0110 -> 6  1010 -> A  1110 -> E
  011 -> 3  111 -> 7       0011 -> 3  0111 -> 7  1011 -> B  1111 -> F
(both upper and lower case A-F are used across different computers and
operating systems).
```

The hex and octal representations of those integers above are easy to translate from the binary counterparts; add 0x to the front for a C-style-language hexadecimal number that the compiler will accept:

```
        1001.1010.1101.0101.1010.0010.1010.1011
    ->     9    A    D    5    A    2    A    B  --> 0x9AD5A2AB
                              (that's 0x in front of the hex number)
```

and

```
        10.011.010.110.101.011.010.001.010.101.011
         2   3   2   6   5   3   2   1   2   5   3 -> 023265321253
                              (that's a numeric '0' in front)
```

Octal is easier to write down quickly, but hexadecimal has the nice properties of breaking easily into bytes (which are pairs of hexadecimal digits).

Some aids for remembering the correspondence of hexadecimal (often called 'hex') digits and their decimal digit counterpart:

- hex 0-9 are the same as decimal digits 0-9

- A is the first one past 9 and is easy to remember as 10
- F is the last one and thus easy to remember as 15
- C is decimal 12 (the only one that you sort of have to memorize)
- All the rest are close to A, C, or F (B is just A+1, D is C+1, E is F-1)

If someone mentions the "third bit" of a number, its best to find out if they mean third bit from the left or from the right and whether they start counting from 0 or 1. A miscommunication can result in real problems laster on (i.e., reversed strings of bits!). This page, for example, starts counting with the rightmost bit being number 1. This page is a gem in that answer 5 counts from the right starting at 1, while answer 8 counts from the right starting at 0.

Almost all modern computers use the left-most bit as the 'sign bit' which signifies a negative integer if its value is 1. Note that identifying the location of the sign bit requires knowledge of precisely how many bits are in a given data type. This number can change over time (i.e., when you recompile on a different computer). Generally, the sizeof() operator will tell you the number of **bytes** (which are generally 8 bits) its argument contains, e.g., sizeof(int) or sizeof (100) will yield 4 on a 32-bit machine. Sometimes one can identify a system-based include file that contains the proper constants if a program must depend on a certain word (integer) length.

### Operating on Binary Numbers in Programs

Sometimes it is handy to work with the bits stored in numbers rather than just treating them as integers. Examples of such times include remembering choices (each bit slot can be a 'yes'/'no' indicator), keeping track of option flags (same idea, really, each bit slot is a 'yes'/'no' indicator for a flag's presence), or keeping track of a number of small integers (e.g., successive pairs of bit slots can remember numbers from 0..3). Of course, occasionally programming tasks actually contain 'bit strings'.

In C/C++ and others, assigning a binary number is easy if you know its octal or hexadecimal representation:

```
i = 0x9AD5A2AB;                 /* hexadecimal: 0x */
```

or

```
i = 023265321253;               /* octal: start with 0 */
```

More often, a pair of single-bit valued integers is combined to create an integer of interest. One might think the statement below would do that:

```
i = 0x10000 + 0x100;
```

and it will -- until the sign bit enters the picture or the same bit is combined twice:

```
i = 0x100 + 0x100;
```

In that case, a 'carry' occurs (0x100 + 0x100 = 0x200 which is probably not the result you want) and then i contains 0x200 instead of 0x100 as probably desired. The 'or' operation -- denoted as '|' in C/C++ and others -- does the right thing. It combines corresponding bits in its two operands using these four rules:

```
0 | 0 -> 0
0 | 1 -> 1
1 | 0 -> 1
1 | 1 -> 1
```

The '|' operation is called 'bitwise or' in C so as not to be confused with it's cousin '||' called 'logical or' or 'orif'. The '||' operator evaluates the arithmetic value of its left side operand and, if that value is false (exactly 0), it evaluates its right side operand. The 'orif' operator is different: if either operand is nonzero, then '||' evaluates to true (exactly 1 in C).

It is "1 | 1 = 1" that distinguishes the '|' operator from '+'. Sometimes operators like this are displayed as a 'truth table':

```
               right operand
operator | | 0  1
        ---+------
   left  0 | 0  1    <-- results
operand  1 | 1  1    <-- results
```

It's easy to see that the 'bitwise or' operation is a way to set bits inside an integer. A '1' results with either or both of the input bits are '1'.

The easy way to query bits is using the 'logical and' (also known as 'bitwise and') operator which is denoted as '&' and has this truth table:

```
 & | 0  1
---+------
 0 | 0  0
 1 | 0  1
```

Do not confuse the single '&' operator with its partner named 'andif' with two ampersands ('&&'). The 'andif' operator will evaluate its left side and yield 0 if the left side is false, without evaluating its right side operand. Only if the left side is true will the right side be evaluated, and the result of the operator is the logical 'and' of their truth values (just as above) and is evaluated to either the integer 0 or the integer 1 (vs. '&' which would yield 4 when evaluating binary 100100 & 000111).

A '1' results only when *both* input bits are '1'. So, if a program wishes to know if the 0x100 bit is '1' in an integer, the if statement is simple:

```
if (a & 0x100) { printf("yes, 0x100 is on\n"); }
```

C/C++ (and others) contain additional operators, including 'exclusive or' (denoted '^') with this truth table:

```
 ^ | 0  1
---+------
 0 | 0  1
 1 | 1  0
```

The 'exclusive or' operator is sometimes called 'xor', for easy of typing. Xor yields a '1' either exactly *one* of its inputs is one: either one or the other, but not both. This operator is very handy for 'toggling' (flipping) bits, changing them from '1' to '0' or vice-versa. Consider this statement:

```
a = a ^ 0x100;   /* same as a ^= 0x100; */
```

The 0x100 bit will be changed from 0->1 or 1->0, depending on its current value.

Switching off a bit requires two operators. The new one is the unary operator that toggles every bit in a word, creating what is called the 'bitwise complement' or just 'complement' of a word. Sometimes this is called 'bit inversion' or just 'inversion' and is denoted by the tilde: '~'. Here's a quick example:

```
char a, b;      /* eight bits, not 32 or 64 */
a = 0x4A;       /* binary 0100.1010 */
b = ~a;         /* flip every bit: 1011.0101 */
printf("b is 0x%X\n", b);
```

which yields:

```
b is 0xB5
```

Thus, if we have a single bit switched on (e.g., 0x100) then ~0x100 has all but one bit switched on: 0xFFFFFEFF (note the 'E' as the third 'digit' from the right) (this example shows a 32-bit value; a 64-bit value would have a lot more F's on the front).

These two operators combine to create a scheme for switching off bits:

```
a = a & (~0x100);       /* switch off the 0x100 bit */
                        /* same as a &= ~0x100;
```

since all but one bit in ~0x100 is on, all the bits except the 0x100 bits appear in the result. Since the 0x100 bit is 'off' in ~0x100, that bit is guaranteed to be '0' in the result. This operation is universally called 'masking' as in 'mask off the 0x100 bit'.

## Summary

In summary, these operators enable setting, clearing, toggling, and testing any bit or combination of bits in an integer:

```
a |= 0x20;              /* turn on bit 0x20 */
a &= ~0x20;             /* turn off bit 0x20 */
a ^= 0x20;              /* toggle bit 0x20 */
if (a & 0x20) {
        /* then the 0x20 bit is on */
}
```

**Shifting**

Moving bits to the left or right is called 'shift'ing. Consider a five-bit binary number like 00110. If that number is shifted one bit left, it becomes: 01100. On the other hand, if 00110 is shift one bit to the right, it becomes 000011. Mathematically inclined users will realize that shifting to the left one bit is the same as multiplying by 2 while shifting to the right one bit is usually the same as an integer divide by 2 (i.e., one discards any remainder). Why usually? Shifting -1 right by one yields an unusual result (i.e,, 0xFFFF.FFFF >> 1 == 0xFFFF.FFFF, no change at all).

Generally, one can specify a shift by more than one bit: Shifting 000001 to the left by three bits yields 001000. The shift operators are:

```
a << n        /* shift a left n bits
a >> n        /* shift a right n bits
```

One generally shifts integers (instead of floating-point numbers), although nothing prevents shifting floating point numbers in some languages. The results are generally very difficult to interpret as a floating point number, of course.

When shifting to the left, 0's are inserted in the lower end.
**When shifting to the right, the high order bit is duplicated and inserted for the newly-needed bit** (thus preserving the number's sign). This means that (-1)>>1 yields -1 instead of 0!

Another type of shift, unavailable natively in most programming languages is the 'circular' shift, where bits shifted off one end are inserted at the other end instead of the default 0. Modern uses of this operation are rare but could appear occasionally. Some machines (e.g., the x86 architecture) have assembly-language instructions for this, but the prudent C or Java programming will spend a few more machine cycles (billionths of a second) to execute both a shift and then a bit-extract-shift-or combination to move the bit themselves.

A note on optimization: some zealous optimizing coders like to change a/4 to '(a>>2)' in order to save time ("since multiplies can be slow"). Modern compilers know all about this and perform such substitutions automatically, thus leaving the better programmer to write a/4 when that is what's meant.

**Very Advanced Bit Manipulation**

**Skip this section if this is your first time dealing with bits. Read it at your leisure in the future after you've written some bit manipulation code. Really.**

It turns out that the way 2's complement machines represent integers and the way they implement subtraction (the standard on virtually all modern machines) yields some very interesting possibilities for bit manipulation.

Two's complement machines represent positive integers as the binary representation of that integer with a 0 in the sign bit. A negative integer is represented as the complement of the positive integer (including turning on the sign bit) plus 1. Thus, the absolute value of the most negative representable integer is one more than the most positive representable integer. Thus:

| x | +x in 8-bit binary | -x in 8-bit binary |
|---|---|---|
| 0 | 0000 0000 | 0000 0000 |
| 1 | 0000 0001 | 1111 1111 |
| 2 | 0000 0010 | 1111 1110 |
| 3 | 0000 0011 | 1111 1101 |
| 64 | 0100 0000 | 1100 0000 |
| 65 | 0100 0001 | 1011 1111 |
| 126 | 0111 1110 | 1000 0010 |
| 127 | 0111 1111 | 1000 0001 |
| 128 | [no representation] | 1000 0000 |

Given this representation, addition can proceed just as on pencil and paper, discard any extra high order bits that exceed the length of the representation. Subtracting b from a (a-b) proceeds as add a and the quantity (-b).

This means that certain bit operations can exploit these definitions

of negation and subtraction to yield interesting results, the proofs of which are left to the reader (see a table of these offsite):

| Value | Binary Sample | Meaning |
|---|---|---|
| x | 00101100 | the original x value |
| x & -x | 00000100 | extract lowest bit set |
| x \| -x | 11111100 | create mask for lowest-set-bit & bits to its left |
| x ^ -x | 11111000 | create mask bits to left of lowest bit set |
| x & (x-1) | 00101000 | strip off lowest bit set<br>--> useful to process words in O(bits set)<br>instead of O(nbits in a word) |
| x \| (x-1) | 00101111 | fill in all bits below lowest bit set |
| x ^ (x-1) | 00000111 | create mask for lowest-set-bit & bits to its right |
| ~x & (x-1) | 00000011 | create mask for bits to right of lowest bit set |
| x \| (x+1) | 00101101 | toggle lowest zero bit |
| x / (x&-x) | 00001011 | shift number right so lowest set bit is at bit 0 |

There's no reason to memorize these expressions, but rather remember what's possible to refer back to this page for saving time when you processing bits.

USACO Gateway | Comment or Question

# Complete Search

## The Idea

Solving a problem using complete search is based on the ``Keep It Simple, Stupid'' principle. The goal of solving contest problems is to write programs that work in the time allowed, whether or not there is a faster algorithm.

Complete search exploits the brute force, straight-forward, try-them-all method of finding the answer. This method should almost always be the first algorithm/solution you consider. If this works within time and space constraints, then do it: it's easy to code and usually easy to debug. This means you'll have more time to work on all the hard problems, where brute force doesn't work quickly enough.

In the case of a problem with only fewer than a couple million possibilities, iterate through each one of them, and see if the answer works.

## Careful, Careful

Sometimes, it's not obvious that you use this methodology.

### Problem: Party Lamps [IOI 98]

You are given N lamps and four switches. The first switch toggles all lamps, the second the even lamps, the third the odd lamps, and last switch toggles lamps 1, 4, 7, 10, ... .

Given the number of lamps, *N*, the number of button presses made (up to 10,000), and the state of some of the lamps (e.g., lamp 7 is off), output all the possible states the lamps could be in.

Naively, for each button press, you have to try 4 possibilities, for a total of $4^{10000}$ (about $10^{6020}$ ), which means there's no way you could do complete search (this particular algorithm would exploit recursion).

Noticing that the order of the button presses does not matter gets this number down to about $10000^4$ (about $10^{16}$ ), still too big to completely search (but certainly closer by a factor of over $10^{6000}$ ).

However, pressing a button twice is the same as pressing the button no times, so all you really have to check is pressing each button either 0 or 1 times. That's only $2^4 = 16$ possibilities, surely a number of iterations solvable within the time limit.

### Problem 3: The Clocks [IOI 94]

A group of nine clocks inhabits a 3 x 3 grid; each is set to 12:00, 3:00, 6:00, or 9:00. Your goal is to manipulate them all to read 12:00. Unfortunately, the only way you can

manipulate the clocks is by one of nine different types of move, each one of which rotates a certain subset of the clocks 90 degrees clockwise.

Find the shortest sequence of moves which returns all the clocks to 12:00.

The ``obvious'' thing to do is a recursive solution, which checks to see if there is a solution of 1 move, 2 moves, etc. until it finds a solution. This would take $9^k$ time, where $k$ is the number of moves. Since $k$ might be fairly large, this is not going to run with reasonable time constraints.

Note that the order of the moves does not matter. This reduces the time down to $k^9$ , which isn't enough of an improvement.

However, since doing each move 4 times is the same as doing it no times, you know that no move will be done more than 3 times. Thus, there are only $4^9$ possibilities, which is only 262,144, which, given the rule of thumb for run-time of more than 10,000,000 operations in a second, should work in time. The brute-force solution, given this insight, is perfectly adequate.

## Sample Problems

#### Milking Cows [USACO 1996 Competition Round]

Given a cow milking schedule (Farmer A milks from time 300 to time 1000, Farmer B from 700 to 1200, etc.), calculate

- The longest time interval in which at least one cow was being milked
- The longest time interval in which no cow is being milked

#### Perfect Cows & Perfect Cow Cousins [USACO 1995 Final Round]

A perfect number is one in which the sum of the proper divisors add up to the number. For example, 28 = 1 + 2 + 4 + 7 + 14. A perfect pair is a pair of numbers such that the sum of the proper divisor of each one adds up to the other. There are, of course, longer perfect sets, such that the sum of the divisors of the first add up to the second, the second's divisors to the third, etc., until the sum of the last's proper divisors add up to the first number.

Each cow in Farmer John's ranch is assigned a serial number. from 1 to 32000. A perfect cow is one which has a perfect number as its serial. A group of cows is a set of perfect cow cousins if their serial numbers form a perfect set. Find all perfect cows and perfect cow cousins.

USACO Gateway | Comment or Question

# Computational Geometry

## Prerequisites

- Graph Theory
- Shortest Path

## Tools

This module discusses several algorithms that calculate various geometric properties, mostly based on only two operations described below: cross product and arctangent.

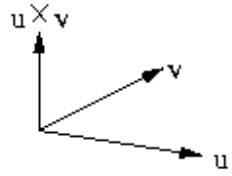
### Cross Product

The cross product of *u* and *v* is written as *u x v*. Computationally, the *cross product* of two three-dimensional vectors *u* and *v* is the vector determinant of the following matrix (where **i**, **j**, and **k** are unit vectors in the *x*, *y*, and *z* directions respectively):

```
| i  j  k  |
| ux uy uz |
| vx vy vz |
```

That equation works out to:

$(u_y v_z - v_y u_z)\mathbf{i} + (u_z v_x - u_x v_z)\mathbf{j} + (u_x v_y - u_y v_x)\mathbf{k}$

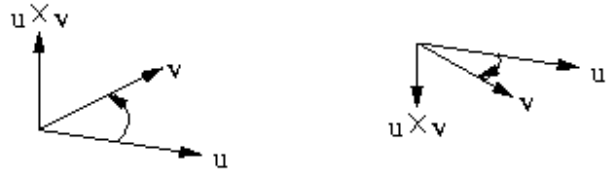



This definition can be used for vectors in two dimensions by using three-dimensional vectors with a z component of 0. The resulting vector will only have a z value.

The cross product has three properties:

- The *cross product* of two vectors is perpendicular to both vectors.
- The length of the cross product is equal to the product of:
  - the length of *u*,
  - the length of *v*, and
  - the sine of the angle between the vectors.

Of the two different directions that are perpendicular to both *u* and *v*, the direction the cross product points depends on whether *u* is ``to the right'' of *v* or ``to the left.''




#### Dot product

The *dot product* of two vectors *u* and *v* is a scalar written as $u \cdot v$. Computationally, it is defined in three dimensions as: $u_x v_x + u_y v_y + u_z v_z$

The dot product is actually equal to the product of:

- the length of *u*
- the length of *v*
- the cosine of the angle between *u* and *v*.

Presuming *u* and *v* are non-zero, if the dot product if negative, *u* and *v* make an angle greater than 90 degrees. If it is zero, then *u* and *v* are perpendicular. If *u cdot v* is positive, then the two vectors form an acute angle.

#### Arctangent

The *arctangent*function calculates the (an) angle whose tangent is its argument and generally returns a real number between -pi/2 and pi/2. An additional function in C, *atan2*, takes two arguments: a *DELTA y* value and a *DELTA x* value (in that order!). It determines the angle between the given vector and the positive *x* axis and returns a value between -pi and pi. This has the advantage of removing concerns about dividing by zero or writing code to repair angles in order to handle the negative *x* cases. The *atan2* function is almost always easier to use than the simpler *atan* function that takes only one argument.

## Particular Debugging Problems

The main problem with geometric problems is that they spawn **a lot** of special cases. Be on the lookout for these special cases and **make sure your program works for all of them**.

Floating point calculations also create a new set of problems. Floating point calculations are rarely precise, as the computer only maintains so many bits (digits) of accuracy: be aware of this. In particular, when checking if two values are equal, check to see if they are within some small tolerance of each other not precisely equal.

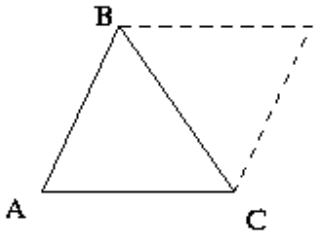
## Geometric Algorithms

Here are some of snippets that can help you solve geometry problems.

#### Area of Triangle

To calculate the area of a triangle with vertices (a, b, c), pick a vertex (say a) and create a vector to the other two vertices (let *u* = b - a, and *v* = c - a). The area of the triangle

(a, b, c) is one half the length of cross product *u x v*.




An alternative method to find the area of triangle is to use Hero's formula. If the lengths of the sides of a triangle are *a*, *b*, and *c*, let *s = (a+b+c)/2*. The area of the triangle is then

*sqrt(s* (s-a)*(s-b)*(s-c))* .

**Are Two Line Segments Parallel?**

To check if two line segments are parallel, create vectors along each line segment and check to see if their cross product is (almost) zero.

**Area of polygon**

The area of a polygon with vertices ($x_1$, $y_1$), ..., ($x_n$, $y_n$) is equal to the determinant:

```
 1   | x1 x2 ... xn |
---  |              |
 2   | y1 y2 ... yn |
```

where the determinate is defined to be similar to the 2 by 2 determinant: $x_1 y_2 + x_2 y_3 + ... + x_n y_1 - y_1 x_2 - y_2 x_3 - ... - y_n x_1$

**Distance from a point to a line**

The distance from a point P to a line AB is given by the magnitude of the cross product. In particular, *d*(P,AB) = |(P - A) x (B - A)| / | B - A| .

To determine the distance from a point P to the plane defined by A, B, and C, let *n* = (B - A) x (C - A). The distance is then give by the following equation: *d*(P,ABC) = (P-A) · n / |n|.

**Points on a line**

A point is on a line if the distance from the point to the line is 0.

**Points on the same side of line**

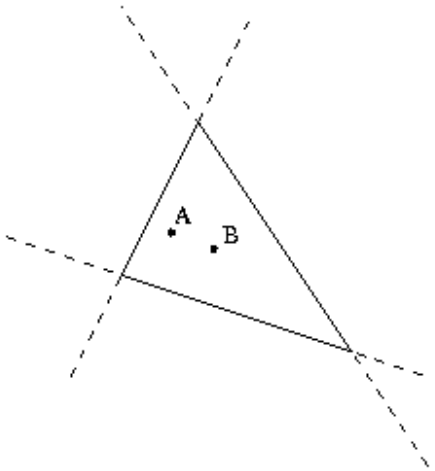
This notion only makes sense for two dimensions. To check if points C and D are on the same side of line AB, calculate the z component of *(B - A) x (C - A)* and *(B - A) x (D - A)*. If the *z* components have the same sign (i.e., their product is positive), then C and D are on the same side of the line AB.

**Point on line segment**

To calculate if a point C is on the line segment AB, check if C is on the line AB. If it is, then check if the length of AB is equal to the sum of the lengths of AC and CB.

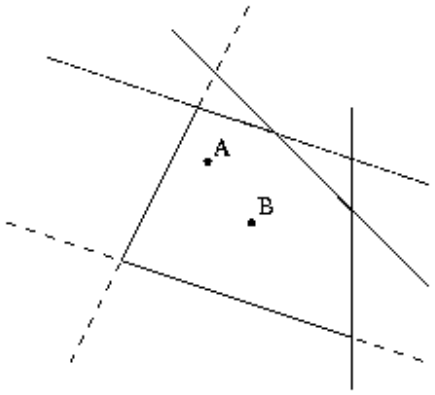
**Point in triangle**

To check if a point A is in a triangle, find another point B which is within the triangle (the average of the three vertices works well). Then, check if the point A is on the same side of the three lines defined by the edges of the triangle as B.




**Point in convex polygon**

The same trick works for a convex polygon:




**Four (or more) points are coplanar**

To determine if a collection of points is coplanar, select three points, A, B, and C. Now, if, for any other point D, $(B - A) \times (C - A)) \cdot (D - A) = \sim 0$, then the collection of points resides in some plane.

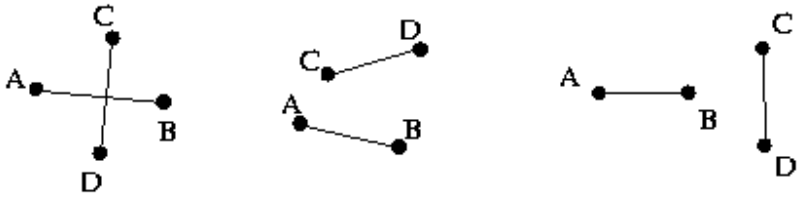
**Two lines intersect**

Two lines intersect if and only if they are not parallel in two dimensions.

In three dimensions, two lines AB and CD intersect if they are not parallel and A, B, C, and D are coplanar.

**Two line segments intersect**

In two dimensions, two line segments AB and CD intersect if and only if A and B are on opposite sides of the line CD and C and D are on opposite sides of line AB.




Note that both of the checks are necessary, as for the last case one of the checks returns true, while the other testifies to the fact that AB and CD do not intersect. In three dimensions, solve following system of equations, where *i* and *j* are the unknowns:

$$A_x + (B_x - A_x)\ i = C_x + (D_x - C_x)\ j$$
$$A_y + (B_y - A_y)\ i = C_y + (D_y - C_y)\ j$$
$$A_z + (B_z - A_z)\ i = C_z + (D_z - C_z)\ j$$

If this system has a solution (*i*, *j*), where $0 <= i <= 1$ and $0 <= j <= 1$, then the line segments intersect at: $(A_x + (B_x - A_x)i,\ A_y + (B_y - A_y)i,\ A_z + (B_z - A_z)\ i$ .

**Point of Intersection of Two Lines**

For the lines AB and CD in two dimensions, the most straight-forward way to calculate the intersection of them is to solve the system of two equations and two unknowns:

$$A_x + (B_x - A_x)i = C_x + (D_x - C_x)\ j$$
$$A_y + (B_y - A_y)i = C_y + (D_y - C_y)\ j$$

The point of intersection is:

$$(A_x + (B_x - A_x)\ i,\ A_y + (B_y - A_y)\ i)$$

In three dimensions, solve the same system of equations as was used to check line intersection, and the point of intersection is:

$$(A_x + (B_x - A_x)i,\ A_y + (B_y - A_y)i,\ A_z + (B_z - A_z)i)$$

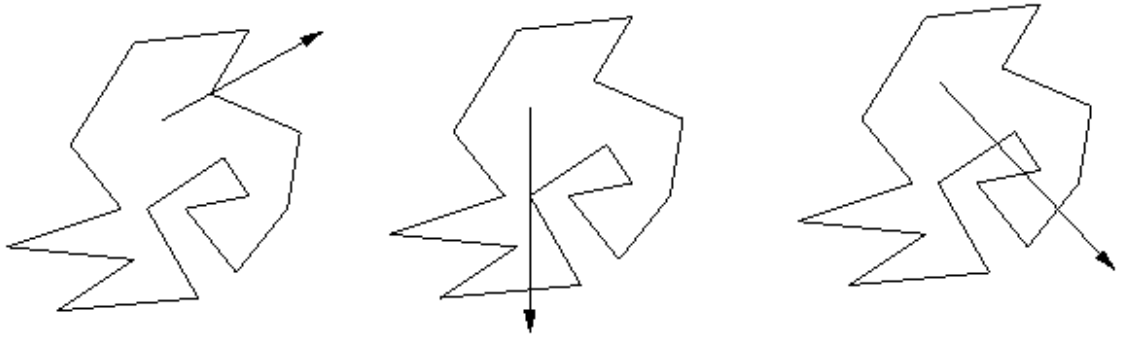
**Checking convexity of 2-dimensional polygon**

To check the convexity of a 2-dimensional polygon, walk the polygon in clock-wise order. For each triplet of consecutive points (A, B, C), calculate the cross product (B - A) x (C - A). If the *z* component of each of these vectors is positive, the polygon is convex.

**Point in non-convex polygon**

To calculate if a point is within a nonconvex polygon, make a ray from that point in a random direction and count the number of times it intersects the polygon. If the ray intersects the polygon at a vertex or along an edge, pick a new direction. Otherwise, the point is within the polygon if and only if the ray intersects the polygon an odd number of

times.



This method also extends to three dimensions (and higher), but the restriction on intersection is that it only intersects at faces and not at either a vertex or an edge.

## Geometry Methodologies

Geometric problems introduce several different tricks that can be used to either reduce the run-time or approximate the solution.

#### Monte Carlo

The first geometric trick is based on randomness. Instead of calculating the probability that something occurs, simulate a random event and calculate the fraction of times it occurs. If enough events are simulated, the difference between these two values becomes very small.

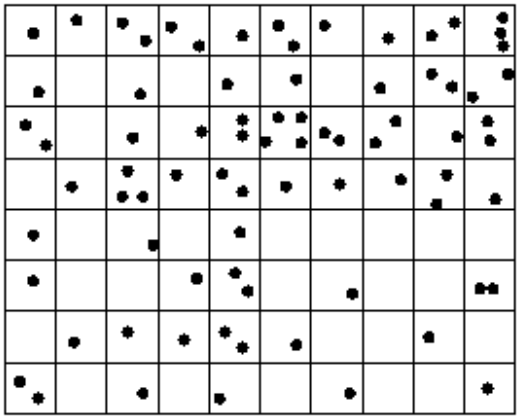
This can be helpful to determine something like the area of a figure. Instead of calculating the area directly, determine a bounding box, and throw ``darts'' at the box, and estimate what the probability of hitting the figure is. If this is calculated accurately enough, this can give a good estimate of the actual area.

The problem with this method is to get a good relative error (error divided by the actual value) requires a large number of successful events. If the probability of the event occurring is very small, the method does not yield good results.

#### Partitioning

Partitioning is a method to improve the speed of a geometric algorithm. This entails dividing the plane up into sections (usually by a grid but sometimes into radial sections or some other method), and bucketing the objects into appropriate section(s). When looking for objects within some figure, only those sections which have a non-zero intersection with that figure need to be examined, thereby greatly reducing the cost of the algorithm. This is helpful to determine the set of objects within some distance of a

given point (the figure is a circle) or to check for intersections (the figure is a line).



#### Graph Problems

Sometimes what may look like a geometric problem is really a graph problem. Just because the input is points in the plane does not mean it's a geometric algorithm.

## Example Problems

#### Point Moving

Given a set of line segments in the plane, and two points A and B, is it possible to move from A to B without crossing any of the segments?

The line segments partition the plane into regions. Determine these regions, and see if A and B reside in the same region.

#### Bicycle Routing

Given a collection of non-intersecting buildings along with start and end locations, find the shortest path from A to B that doesn't go through any buildings.

Analysis: This is really a graph problem. The nodes are the start and end locations, along with the vertices of the buildings. There are edges between any two nodes such that the line segment between them does not intersect any buildings, with weight equal to the length of the length of the line segments. Once that graph has been calculated, the problem is shortest path.

#### Maximizing Line Intersections

Given a collection of segments in the plane, find the greatest number of segments which can by intersected by drawing a single line.

Analysis: With a little bit of thought, it is clear that the line segment must pass through two of the vertices of the collection of line segments. Thus, try all pairs of vertices, and calculate the crossing for each. Combining this with partitioning gives an algorithm that runs fairly quickly.

#### Polygon Classification

Given a collection of segments defining a polygon, determine if it is simple (no two non-consecutive line segments intersect) and convex.

USACO Gateway | Comment or Question

# Crafting Winning Solutions

A good way to get a competitive edge is to write down a game plan for what you're going to do in a contest round. This will help you script out your actions, in terms of what to do both when things go right and when things go wrong. This way you can spend your thinking time in the round figuring out programming problems and not trying to figure out what the heck you should do next... it's sort of like precomputing your reactions to most situations.

Mental preparation is also important.

**Game Plan For A Contest Round**

Read through ALL the problems FIRST; sketch notes with algorithm, complexity, the numbers, data structs, tricky details, ...

- Brainstorm many possible algorithms - then pick the stupidest that works!
- DO THE MATH! (space & time complexity, and plug in actual expected and worst case numbers)
- Try to break the algorithm - use special (degenerate?) test cases
- Order the problems: shortest job first, in terms of your effort (shortest to longest: done it before, easy, unfamiliar, hard)

Coding a problem - For each, one at a time:

- Finalize algorithm
- Create test data for tricky cases
- Write data structures
- Code the input routine and test it (write extra output routines to show data?)
- Code the output routine and test it
- Stepwise refinement: write comments outlining the program logic
- Fill in code and debug *one section at a time*
- Get it working & verify correctness (use trivial test cases)
- Try to break the code - use special cases for code correctness
- Optimize progressively - only as much as needed, and keep all versions (use hard test cases to figure out actual runtime)

**Time management strategy and "damage control" scenarios**

Have a plan for what to do when various (foreseeable!) things go wrong; imagine problems you might have and figure out how you want to react. The central question is: "When do you spend more time debugging a program, and when do you cut your losses and move on?". Consider these issues:

- How long have you spent debugging it already?
- What type of bug do you seem to have?
- Is your algorithm wrong?
- Do you data structures need to be changed?

- Do you have any clue about what's going wrong?
- A short amount (20 mins) of debugging is better than switching to anything else; but you might be able to solve another from scratch in 45 mins.
- When do you go back to a problem you've abandoned previously?
- When do you spend more time optimizing a program, and when do you switch?
- Consider from here out - forget prior effort, focus on the future: how can you get the most points in the next hour with what you have?

Have a checklist to use before turning in your solutions:

Code freeze five minutes before end of contest?

- Turn asserts off.
- Turn off debugging output.

## Tips & Tricks

- Brute force it when you can
- KISS: Simple is smart!
- Hint: focus on *limits* (specified in problem statement)
- Waste memory when it makes your life easier (if you can get away with it)
- Don't delete your extra debugging output, comment it out
- Optimize progressively, and only as much as needed
- Keep all working versions!
- Code to debug:
  - whitespace is good,
  - use meaningful variable names,
  - don't reuse variables,
  - stepwise refinement,
  - COMMENT BEFORE CODE.
- Avoid pointers if you can
- Avoid dynamic memory like the plague: statically allocate everything.
- Try not to use floating point; if you have to, put tolerances in everywhere (never test equality)
- Comments on comments:
  - Not long prose, just brief notes
  - Explain high-level functionality: ++i; /* increase the value of i by */ is worse than useless
  - Explain code trickery
  - Delimit & document functional sections
  - As if to someone intelligent who knows the problem, but not the code
  - Anything you had to think about
  - Anything you looked at even once saying, "now what does that do again?"
  - Always comment order of array indices
- Keep a log of your performance in each contest: successes, mistakes, and what you could have done better; use this to rewrite and improve your game plan!

## Complexity

#### Basics and order notation

The fundamental basis of complexity analysis revolves around the notion of ``big oh'' notation, for instance: O(*N*). This means that the algorithm's execution speed or

memory usage will double when the problem size doubles. An algorithm of $O(N^2)$ will run about four times slower (or use 4x more space) when the problem size doubles. Constant-time or space algorithms are denoted O(1). This concept applies to time and space both; here we will concentrate discussion on time.

One deduces the O() run time of a program by examining its loops. The most nested (and hence slowest) loop dominates the run time and is the only one mentioned when discussing O() notation. A program with a single loop and a nested loop (presumably loops that execute *N* times each) is $O(N^2)$, even though there is also a O(*N*) loop present.

Of course, recursion also counts as a loop and recursive programs can have orders like $O(b^N)$, O(*N!*), or even $O(N^N)$.

**Rules of thumb**

- When analyzing an algorithm to figure out how long it might run for a given dataset, the first rule of thumb is: modern (2004) computers can deal with 100M actions per second. In a five second time limit program, about 500M actions can be handled. Really well optimized programs might be able to double or even quadruple that number. Challenging algorithms might only be able to handle half that much. Current contests usually have a time limit of 1 second for large datasets.
- 16MB maximum memory use
- $2^{10}$ ~approx~ $10^3$
- If you have *k* nested loops running about *N* iterations each, the program has $O(N^k)$ complexity.
- If your program is recursive with *b* recursive calls per level and has *l* levels, the program $O(b^l)$ complexity.
- Bear in mind that there are *N!* permutations and $2^n$ subsets or combinations of *N* elements when dealing with those kinds of algorithms.
- The best times for sorting *N* elements are O(*N* log *N*).
- **DO THE MATH!** Plug in the numbers.

**Examples**

A single loop with *N* iterations is O(*N*):

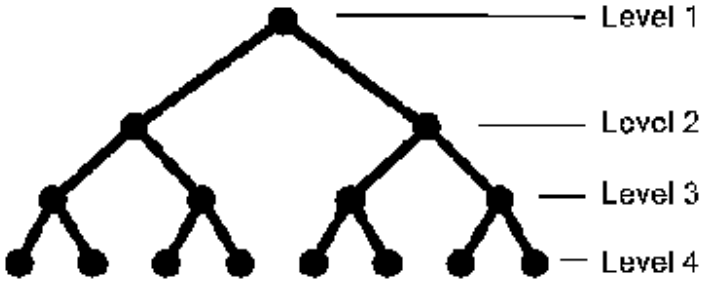
```
 1  sum = 0
 2  for i = 1 to n
 3    sum = sum + i
```

A double nested loop is often $O(N^2)$:

```
# fill array a with N elements
1 for i = 1 to n-1
2   for j = i + 1 to n
3     if (a[i] > a[j])
         swap (a[i], a[j])
```

Note that even though this loop executes *N x (N+1) / 2* iterations of the if statement, it is $O(N^2)$ since doubling *N* quadruples the execution times.

Consider this well balanced binary tree with four levels:




An algorithm that traverses a general binary tree will have complexity $O(2^N)$.

## Solution Paradigms

#### Generating vs. Filtering

Programs that generate lots of possible answers and then choose the ones that are correct (imagine an 8-queen solver) are *filters*. Those that hone in exactly on the correct answer without any false starts are *generators*. Generally, filters are easier (faster) to code and run slower. Do the math to see if a filter is good enough or if you need to try and create a generator.

#### Precomputation

Sometimes it is helpful to generate tables or other data structures that enable the fastest possible lookup of a result. This is called *precomputation* (in which one trades space for time). One might either compile precomputed data into a program, calculate it when the program starts, or just remember results as you compute them. A program that must translate letters from upper to lower case when they are in upper case can do a very fast table lookup that requires no conditionals, for example. Contest problems often use prime numbers - many times it is practical to generate a long list of primes for use elsewhere in a program.

#### Decomposition (The Hardest Thing At Programming Contests)

While there are fewer than 20 basic algorithms used in contest problems, the challenge of combination problems that require a combination of two algorithms for solution is daunting. Try to separate the cues from different parts of the problem so that you can combine one algorithm with a loop or with another algorithm to solve different parts of the problem independently. Note that sometimes you can use the same algorithm twice on different (independent!) parts of your data to significantly improve your running time.

#### Symmetries

Many problems have symmetries (e.g., distance between a pair of points is often the same either way you traverse the points). Symmetries can be 2-way, 4-way, 8-way, and more. Try to exploit symmetries to reduce execution time.

For instance, with 4-way symmetry, you solve only one fourth of the problem and then write down the four solutions that share symmetry with the single answer (look out for self-symmetric solutions which should only be output once or twice, of course).

#### Forward vs. Backward

Surprisingly, many contest problems work far better when solved backwards than when using a frontal attack. Be on the lookout for processing data in reverse order or building an attack that looks at the data in some order or fashion other than the obvious.

**Simplification**

Some problems can be rephrased into a somewhat different problem such that if you solve the new problem, you either already have or can easily find the solution to the original one; of course, you should solve the easier of the two only. Alternatively, like induction, for some problems one can make a small change to the solution of a slightly smaller problem to find the full answer.

USACO Gateway | Comment or Question

# Data Structures

## Prerequisite

- Graph Theory

## How to pick the perfect data structure

There are several aspect of data structures to consider when selecting the proper way to represent the data for a problem.

#### Will it work?

If the data struct won't work, it's not helpful at all. Ask yourself what questions the algorithm will need to be able to ask the data structure, and make sure the data structure can handle it. If not, then either more data must be added to the structure, or you need to find a different representation.

#### Can I code it?

If you don't know or can't remember how to code a given data structure, pick a different one. Make sure that you have a good idea how each of the operations will affect the structure of the data.

Another consideration here is memory. Will the data structure fit in the available memory? If not, compact it or pick a new one. Otherwise, it is already clear from the beginning that it won't work.

#### Can I Code It In Time?

As this is a timed contest, you have three to five programs to write in five hours. If it'll take you an hour and a half to code just the data structure for the first problem, then you're almost certainly looking at the wrong structure.

#### Can I Debug It?

It is easy to forget this particular aspect of data structure selection. Remember that a program is useless unless it works. Don't forget that debugging time is a large portion of the contest time, so include its consideration in calculating coding time.

What makes a data structure easy to debug? That is basically determined by the following two properties.

- **State Is Easy To Examine** The smaller, more compact the representation, in general, the easier it is to examine. Also, statically allocated arrays are **much** easier to examine than linked lists or even dynamically allocated arrays.

- **State can Be Displayed Easily** For the more complex data structures, the easiest way to examine them is to write a small routine to output the data. Unfortunately, given time constraints, you'll probably want to limit yourself to text output. This means that structures like trees and graphs are going to be difficult to examine.

#### Is It Fast?

This is, surprisingly, the least important consideration when picking a data structure. A slow program will normally get a noticeable portion of the points, but a fast, incorrect one will not, unless it gets lucky.

#### Conclusion

In general, remember the KISS principle: ``Keep It Simple, Stupid.'' Sometimes more complexity is very helpful, but make sure you're getting your money's worth. Remember that taking the time to make sure you've got the correct data structure at the start is a lot less expensive than having to replace a data structure later.

## Things to Avoid: Dynamic Memory

In general, you should avoid dynamic memory, because:

### It Is Too Easy To Make Mistakes Using Dynamic Memory

Overwriting past allocated memory, not freeing memory, and not allocating memory are only some of the mistakes that are introduced when dynamic memory is used. In addition, the failure modes for these errors are such that it's hard to tell where the error occurred, as it's likely to be at a (potentially much later) memory operation.

### It Is Too Hard To Examine the Data Structure's Contents

The interactive development environments available don't handle dynamic memory well, especially for C.

Consider parallel arrays as an alternative to dynamic memory. One way to do a linked list, where instead of keeping a next point, you keep a second array, which has the index of the next element. Sometimes you may have to dynamically allocate these, but as it should only be done once, it's much easier to get right than allocating and freeing the memory for each insert and delete.

All of this notwithstanding, sometimes dynamic memory is the way to go, especially for large data structures where the size of the structure is not known until you have the input.

## Things to Avoid: Coolness Factor

Try not to fall into the ``coolness'' trap. You may have just seen the neatest data structure, but remember:

- **Cool ideas that don't work aren't.**
- **Cool ideas that'll take forever to code aren't, either**

It's much more important that your data structure and program work than how impressive your data structure is.

## Basic Structures

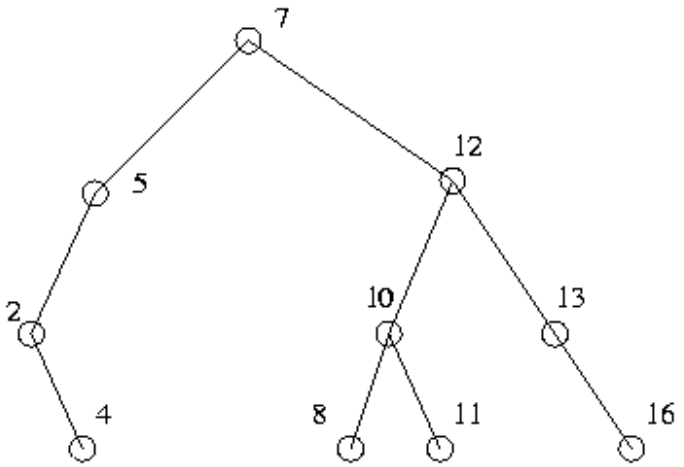
There are five basic data structures: arrays, linked lists, stacks, queues, and deque (pronounced deck). You might have seen these structures before. If you have not, consult Sedgewick for what they are.

#### Binary Search Trees

Binary search trees enable you to search a collection of objects (each with a real or integer value) quickly to determine if a given value exists in the collection.

Basically, a binary search tree is a weighted, rooted binary ordered tree. That collection of adjectives means that each node in the tree might have one `right' child and one `left' child (but both or either could be missing). In addition, each node has an object associated with it, and the `weight' of the node is the value of the object.

The binary search tree also has the property that each node's left child and descendants of its left child have a value less than that of the node, and each node's right child and its descendants have a value greater or equal to it.




The nodes are generally represented as a structure with four fields, a pointer to the node's left child, a pointer to the node's right child, the weight of the object stored at this node, and a pointer to the object itself.

#### Why Are Binary Search Trees Useful?

Given a collection of N objects, a binary search tree takes only O(log N) time to find an objects, assuming that the tree is not really poor (a tree where each node has no left child means the search will take O(N) time, for example). In addition, unlike just keeping a sorted array, inserting and deleting objects only takes log N time as well.

#### Variations on Binary Trees

Sometimes it's helpful to provide a link to a node's parent as well.

There are several variants that ensure that the trees are never poor. Splay trees, Red-black trees, Treaps, B-trees, and AVL trees are some of the more common examples. They are all much more complicated to code, and random trees are generally good, so it's generally not worth it.

If you're concerned that the tree you created might be bad (it's being created by inserting elements from an input file, for example), then randomly order the elements before insertion.

**Hash Tables**

A hash table stores data with a very quick way to do lookups. Let's say there is a collection of objects and a data structure must quickly answer the question: ``Is this object in the data structure?'' (e.g., is this word in the dictionary?). A hash table does this in less time than it takes to do binary search.

The idea is this: find a function that maps the elements of the collection to an integer between 1 and x (where x, in this explanation, is larger than the number of elements in your collection). Keep an array indexed from 1 to x, and store each element at the position that the function evaluates the element as. Then, to determine if something is in your collection, just plug it into the function and see whether or not that position is empty. If it is not check the element there to see if it is the same as the something you're holding,

For example, presume the function is defined over 3-character words, and is (first letter + (second letter * 3) + (third letter * 7)) mod 11 (A=1, B=2, etc.), and the words are ``CAT'', ``CAR'', and ``COB''. When using ASCII, this function takes ``CAT'' and maps it to 3, maps ``CAR'' to 0, and maps ``COB'' to 7, so the hash table would look like this:

```
0: CAR
1
2
3: CAT
4
5
6
7: COB
8
9
10
```

Now, to see if ``BAT'' is in the table, plug it into the hash function to get 2. This position in the hash table is empty, so it is not in the collection. ``ACT'', on the other hand, returns the value 7, so the program must check to see if that entry, ``COB'', is the same as ``ACT''.

Consider this function:

```
#define NHASH 8999          /* make sure this is prime! */

hashnum(p)
char *p;
{
    unsigned int sum = 0;
    for ( ; *p; p++)
        sum = (sum << 3) + *p;
    return sum % NHASH;
}
```

This function will return *some* integer in the range 0..NHASH-1 for every input. As it turns out, the output is fairly random. this simple function for NHASH to be prime. Combine the above with a main program:

```
#include

main() {
    FILE *in;
    char line[100], *p;
    in = fopen ("/usr/share/dict/words", "r");
    while (fgets (line, 100, in)) {
        for (p = line; *p; p++)
            if (*p == '\n') { *p = '\0'; break; }
        printf("%6d %s\n", hashnum(line), line);
    }
    exit (0);
}
```

to yield numbers like this for the (start of the) english dictionary:

```
  4645 aback
  4678 abaft
  6495 abandon
  2634 abandoned
  4810 abandoning
   142 abandonment
  7080 abandons
  4767 abase
  2240 abased
  7076 abasement
  4026 abasements
  2255 abases
  4770 abash
   222 abashed
   237 abashes
  2215 abashing
   361 abasing
  4775 abate
  2304 abated
  3848 abatement
   ... ...
```

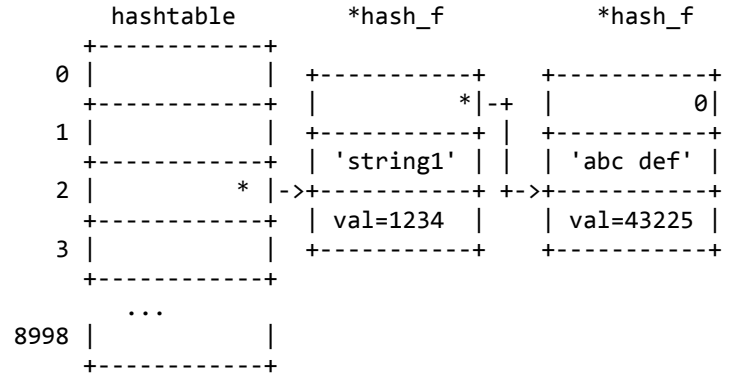
You can see that the function yields numbers that are all different and are fairly random looking, at least in this small sample.

Of course, if one has NHASH+1 words, the pigeon-hole principle says that at leaast one pair of them will yield the same function value. This is called a 'collision'. Pragmatic hash tables use a list of length NHASH to represent the head of NHASH linked lists of words that all hashed to the same value.

Let's see how hashing is really used. First, start with a structure that forms a linked list off the hash table. The linked list structure looks like this:

```
struct hash_f {
    struct hash_f *h_next;
    char *h_string;
    int   h_value;   /* some value associated with the string */
                /* completely optional how it's used or even if it's present */
};

struct hash_f *hashtable[NHASH];      /* the head of each linked list */
                          /* automatically set to NULL since it's global */
```

This makes a hash table that would look like this if two elements were present:




Here's hashinsert:

```
struct hash_f *
hashinsert(p, val)
char *p;
int val;
{
    int n = hashnum(p);                         /* where in table? */
    struct hash_f *h = malloc( sizeof (struct hash_f) ); /* make a new hash element */

/* link into start of list: */
    h->h_next = hashtable[n];
    hashtable[n] = h;

/* optional value: */
    h->h_val = val;

/* so we can later find the proper element in this chain: */
    h->h_string = malloc( strlen(p) + 1 );
    strcpy (h->h_string, p);

    return h;
}
```

And here's hashlookup (which will return a pointer to the hash structure if it's found):

```
struct hash_f *
hashlookup(p) {
    struct hash_f *h;

    int n = hashnum(p);                         /* where to start looking */

    for (h = hashtable[n]; h; h=h->h_next)      /* traverse linked list */
        if (0 == strcmp (p, h->h_string))       /* string match? done! */
            return h;
    return 0;                                   /* didn't find target */
}
```

Now you can insert strings quickly and look them up quickly, in size_of_linked_list/2 string compares, on average.

**Why Are Hash Tables Useful?**

Hash tables enable, with a little bit of memory cost, programs to perform lookups with almost constant work. Generally, the program must evaluate the function and then possibly compare the looked up element to an entry or a few entries in the table.

**Hashers**

A more subtle, and often forgotten, technique to avoid collisions is to pick a good hash function. For example, taking the three letter prefix as the hash value for a dictionary would be very bad. Under this hash function, the prefix ``CON'' would have a huge number of entries. Pick a function where two elements are unlikely to map to the same value:

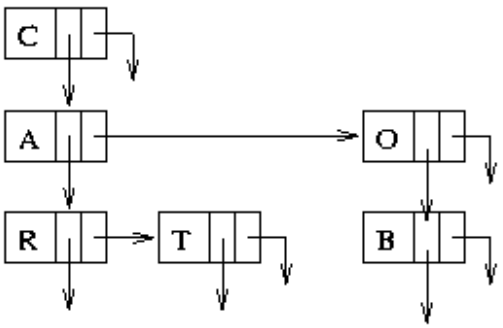
- Create a relatively huge value and mod it with the size of your table (this works especially well if your hash table is a prime size).
- Primes are your friends. Multiply by them.
- Try to have small changes map to completely different locations.
- You don't want to have two small changes cancel each other out in your mapping function (a transposition, for example).
- This is a whole field of study, and you could create a ``Perfect Hash Function'' that would give you no collisions, but, for your purposes, that's entirely too much work. Pick something that seems fairly random, and see if it works; it probably will.

**Hash Table Variations**

It is often quite useful to store more information that just the value. One example is when searching a small subset of a large subset, and using the hash table to store locations visited, you may want the value for searching a location in the hash table with it.

Even a small hash table can improve runtime by drastically reducing your search space. For example, keeping a dictionary hashed by the first letter means that if you wanted to search for a word, you would only be looking at words that have the same first letter.

## Special Trees Called 'Tries'




A trie is, in short, a rooted tree. It has unbounded out-degree (a node may logically have any number of children). The children of a node are stored in a linked list, so each node has two pointers, next sibling and first child.

Tries store a collection of sequences. Every path from the root to a leaf specifies an element of that collection. For example, for the trie illustrated, the collection specified is ``CAR'', ``CAT'', and ``COB'', presuming no other nodes exist.

To determine if a sequence is in the collection, start at the root, search through its children for the first element of that sequence. If no match is found, the sequence is not

in the collection. Otherwise, search the children of that node similarly and for subsequent element.

Tries sport several nice features. Checking to see if an element is in the list takes time bounded by the length of the sequence times the maximum number of children a node has. Additionally, this data structure can often use less memory than other representations, because prefixes only appear once (in our example, there is only one `CA' node even though `CAR' and `CAT' appear).

In general, a trie is nice to use when you want to ask this question a lot: Does there exist a sequence (word, multi-digit number, or other type) that starts with this?

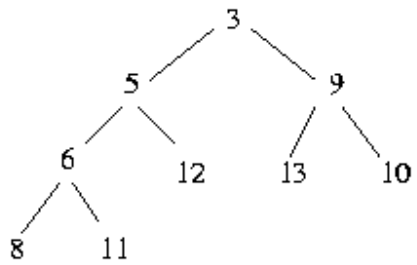
#### Trie Variations

Some common slight alterations to tries are:

- If your list may contain two words, one of which is a prefix of the other, you have to add a flag at each node which says ``A word ends here.'' For example, if you wanted to also have ``CA'' in your trie, it would have to be flagged.
- It sometimes helps to keep the linked list of children in sorted order. This increases the time to build the trie, but decreases the time to search it.
- If you have many words with similar prefixes but unique endings, it is sometimes helpful to put ``special'' nodes that stand for a sequence themselves. For example, if you want to represent ``CARTING,'' ``COBBLER,'' and ``CATCHING,'' memory constraints may make it more reasonable to represent a ``RTING,'' ``BBLER,'' and ``TCHING'' node. Note that this increases the complexity dramatically.

### Heap

A heap (sometimes called a priority queue) is a complete binary tree in which every node's value is less than both of its children's values:




#### Heap Representation

If the tree is filled in from left to right level by level (that is, the tree is complete except for its lowest level, on which the elements are filled in from left to right), then the heap can be stored in an array, which is just the top level from left to right, the second level, and so on.

The heap given would be:

```
3 5 9 6 12 13 10 8 11
```

In this representation, the children of the node at position x are 2x and 2x+1 (assuming 1 based indexing), and the parent of x is truncate(x/2).

**How Do I Add And Remove things To A Heap?**

To add an element, put it at the end of the array. Now, as long as it's below its current parent, swap it with its parent. For example, to add the number 4, the heap array would go through the following states:

```
 3  5  9  6 12 13 10  8 11  4
 3  5  9  6  4 13 10  8 11 12
 3  4  9  6  5 13 10  8 11 12
```

Deleting an element is also relatively easy. Take the last element in the array and replace the element you wish to delete with it. While one of its children is less than it, pick the smaller of the two and swap with it. For example, to delete 3:

```
11  5  9  6 12 13 10  8
 5 11  9  6 12 13 10  8
 5  6  9 11 12 13 10  8
 5  6  9  8 12 13 10 11
```

**What If I Want To Alter A Value?**

To alter a value upwards, change the value, and swap with its parents as long as necessary.

To alter a value downwards, change the value, and swap with the smaller of its two children as long as necessary.

**Why Would I Use This?**

A heap makes it very easy to ask the question ``What's the smallest thing?'' in a collection of dynamic values. It's a compact representation and quick to compute. An example of a location where this can be helpful is Dijkstra's algorithm.

**Heap Variations**

In this representation, just the weight was kept. Usually you want more data than that, so you can either keep that data and move it (if it's small) or keep pointers to the data.

Since when you want to fiddle with values, the first thing you have to do is find the location of the value you wish to alter, it's often helpful to keep that data around. (e.g., node x is represented in location 16 of the heap).

USACO Gateway | Comment or Question

# Dynamic Programming

## Introduction

Dynamic programming is a confusing name for a programming technique that dramatically reduces the runtime of algorithms: from exponential to polynomial. The basic idea is to try to avoid solving the same problem or subproblem twice. Here is a problem to demonstrate its power:

Given a sequence of as many as 10,000 integers (0 < integer < 100,000), what is the maximum decreasing subsequence? Note that the subsequence does not have to be consecutive.

## Recursive Descent Solution

The obvious approach to solving the problem is recursive descent. One need only find the recurrence and a terminal condition. Consider the following solution:

```
 1 #include <stdio.h>


 2  long n, sequence[10000];
 3  main () {
 4       FILE *in, *out;
 5       int i;
 6       in = fopen ("input.txt", "r");
 7       out = fopen ("output.txt", "w");
 8       fscanf(in, "%ld", &n);
 9       for (i = 0; i < n; i++) fscanf(in, "%ld", &sequence[i]);
10       fprintf (out, "%d\n", check (0, 0, 999999));
11       exit (0);
12  }


13  check (start, nmatches, smallest) {
14      int better, i, best=nmatches;
15      for (i = start; i < n; i++) {
16          if (sequence[i] < smallest) {
17              better = check (i, nmatches+1, sequence[i]);
18              if (better > best) best = better;
19          }
20      }
21      return best;
22  }
```

Lines 1-9 and and 11-12 are arguably boilerplate. They set up some standard variables and grab the input. The magic is in line 10 and the recursive routine `check'. The `check' routine knows where it should start searching for smaller integers, the length of the longest sequence so far, and the smallest integer so far. At the cost of an extra call, it terminates `automatically' when `start' is no longer within proper range. The `check' routine is simplicity itself. It traverses along the list looking for a smaller integer than the `smallest' so far. If found, `check' calls itself recursively to find more.

The problem with the recursive solution is the runtime:

```
 N   Seconds
 60   0.130
 70   0.360
```

```
 80   2.390
 90  13.190
```

Since the particular problem of interest suggests that the maximum length of the sequence might approach six digits, this solution is of limited interest.

**Starting At The End**

When solutions don't work by approaching them `forwards' or `from the front', it is often fruitful to approach the problem backward. In this case, that means looking at the end of the sequence first.

Additionally, it is often fruitful to trade a bit of storage for execution efficiency. Another program might work from the end of the sequence, keeping track of the longest descending (sub-)sequence so far in an auxiliary variable.

Consider the sequence starting at the end, of length 1. Any sequence of length 1 meets all the criteria for a longest sequence. Notate the `bestsofar' array as `1' for this case.

Consider the last two elements of the sequence. If the penultimate number is larger than the last one, then the `bestsofar' is 2 (which is 1 + `bestsofar' for the last number). Otherwise, it's `1'.

Consider any element prior to the last two. Any time it's larger than an element closer to the end, its `bestsofar' element becomes at least one larger than that of the smaller element that was found. Upon termination, the largest of the `bestsofar's is the length of the longest descending subsequence.

This is fairly clearly an O($N^2$) algorithm. Check out its code:

```
 1  #include <stdio.h>
 2  #define MAXN 10000
 3  main () {
 4      long num[MAXN], bestsofar[MAXN];
 5      FILE *in, *out;
 6      long n, i, j, longest = 0;
 7      in = fopen ("input.txt", "r");
 8      out = fopen ("output.txt", "w");
 9      fscanf(in, "%ld", &n);
10      for (i = 0; i < n; i++) fscanf(in, "%ld", &num[i]);
11      bestsofar[n-1] = 1;
12      for (i = n-1-1; i >= 0; i--) {
13          bestsofar[i] = 1;
14          for (j = i+1; j < n; j++) {
15              if (num[j] < num[i] && bestsofar[j] >= bestsofar[i])  {
16                  bestsofar[i] = bestsofar[j] + 1;
17                  if (bestsofar[i] > longest) longest = bestsofar[i];
18              }
19          }
20      }
21      fprintf(out, "bestsofar is %d\n", longest);
22      exit(0);
23  }
```

Again, lines 1-10 are boilerplate. Line 11 sets up the end condition. Lines 12-20 run the O($N^2$) algorithm in a fairly straightforward way with the `i' loop counting backwards and the `j' loop counting forwards. One line longer then before, the runtime figures show better performance:

```
   N    Secs
1000  0.080
2000  0.240
3000  0.550
4000  0.950
5000  1.450
6000  2.080
```

```
 7000  2.990
 8000  3.700
 9000  4.700
10000  6.330
11000  7.350
```

The algorithm still runs too slow (for competitions) at N=9000.

That inner loop (``search for any smaller number'') begs to have some storage traded for it.

A different set of values might best be stored in the auxiliary array. Implement an array `bestrun' whose index is the length of a long subsequence and whose value is the first (and, as it turns out, `best') integer that heads that subsequence. Encountering an integer larger than one of the values in this array means that a new, longer sequence can potentially be created. The new integer might be a new `best head of sequence', or it might not. Consider this sequence:

```
10 8 9 4 6 3
```

Scanning from right to left (backward to front), the `bestrun' array has but a single element after encountering the 3:

```
0:3
```

Continuing the scan, the `6' is larger than the `3', so the `bestrun' array grows:

```
0:3
1:6
```

The `4' is not larger than the `6', though it is larger than the `3', so the `bestrun' array changes:

```
0:3
1:4
```

The `9' extends the array, since 9>4 and now there's a sequence of length 3 whose 'first' element is 9:

```
0:3
1:4
2:9
```

The `8' changes the array similar to the earlier case with the `4'; since 8<9, the 8 replaces the 9 for a subsequence of length 3 whose 'first' element is (now) 8:

```
0:3
1:4
2:8
```

The `10' extends the array again since 10 > 8 and now there's a longest subsequence of length 4 (10, 8, 4, 3 -- among others):

```
0:3
1:4
2:8
3:10
```

and yields the answer: 4 (four elements in the array).

Because the `bestrun' array probably grows much less quickly than the length of the processed sequence, this algorithm probabalistically runs much faster than the previous one. In practice, the speedup is large. Here's a coding of this algorithm:

```
 1  #include <stdio.h>
 2  #define MAXN 200000
 3  main () {
 4      FILE *in, *out;
 5      long num[MAXN], bestrun[MAXN];
 6      long n, i, j, highestrun = 0;
 7      in = fopen ("input.txt", "r");
 8      out = fopen ("output.txt", "w");
 9      fscanf(in, "%ld", &n);
10      for (i = 0; i < n; i++) fscanf(in, "%ld", &num[i]);
11      bestrun[0] = num[n-1];
12      highestrun = 1;
13      for (i = n-1-1; i >= 0; i--) {
```

```
14        if (num[i] < bestrun[0]) {
15            bestrun[0] = num[i];
16            continue;
17        }
18        for (j = highestrun - 1; j >= 0; j--) {
19            if (num[i] > bestrun[j]) {
20                if (j == highestrun - 1 || num[i] < bestrun[j+1]){
21                    bestrun[++j] = num[i];
22                    if (j == highestrun) highestrun++;
23                    break;
24                }
25            }
26        }
27      }
28      printf("best is %d\n", highestrun);
29      exit(0);
30  }
```

Again, lines 1-10 are boilerplate. Lines 11-12 are initialization. Lines 14-17 are an optimization for a new `smallest' element. They could have been moved after line 26. Mostly, these lines only effect the `worst' case of the algorithm when the input is sorted `badly'.

Lines 18-26 are the important ones that search the bestrun list; these lines contain all the exceptions and tricky cases (bigger than first element? insert in middle? extend the array?). You should try to code this right now -- without memorizing it.

The speeds are impressive. The table below compares this algorithm with the previous one, showing this algorithm worked for N well into five digits:

| N | orig | Improved |
|---|---|---|
| 1000 | 0.080 | 0.030 |
| 2000 | 0.240 | 0.030 |
| 3000 | 0.550 | 0.050 |
| 4000 | 0.950 | 0.060 |
| 5000 | 1.450 | 0.080 |
| 6000 | 2.080 | 0.090 |
| 7000 | 2.990 | 0.110 |
| 8000 | 3.700 | 0.130 |
| 9000 | 4.700 | 0.140 |
| 10000 | 6.330 | 0.160 |
| 11000 | 7.350 | 0.170 |
| 20000 | | 0.290 |
| 40000 | | 0.570 |
| 60000 | | 0.910 |
| 80000 | | 1.290 |
| 100000 | | 2.220 |

Marcin Mika points out that you can simplify this algorithm to this tiny little solution:

```
#include <stdio.h>
#define SIZE 200000
#define MAX(x,y) ((x)>(y)?(x):(y))

int     best[SIZE];        // best[] holds values of the optimal sub-sequence

int
main (void) {
    FILE *in  = fopen ("input.txt", "r");
    FILE *out = fopen ("output.txt", "w");
    int     i, n, k, x, sol = -1;

    fscanf (in, "%d", &n);      // N = how many integers to read in
    for (i = 0; i < n; i++) {
        best[i] = -1;
        fscanf (in, "%d", &x);
        for (k = 0; best[k] > x; k++)
            ;
        best[k] = x;
        sol = MAX (sol, k + 1);
    }
```

```
    printf ("best is %d\n", sol);
    return 0;
}
```

*Not to be outdone, Tyler Lu points out the program below.* The solutions above use a linear search to find the appropriate location in the 'bestrun' array to insert an integer. However, because the auxiliary array is sorted, using binary search will make it run even faster, decreasing the runtime to O(N log N).

```
#include <stdio.h>
#define SIZE 200000
#define MAX(x,y) ((x)>(y)?(x):(y))

int     best[SIZE];        // best[] holds values of the optimal sub-sequence

int
main (void) {
    FILE *in  = fopen ("input.txt", "r");
    int i, n, k, x, sol;
    int low, high;

    fscanf (in, "%d", &n);      // N = how many integers to read in
    // read in the first integer
    fscanf (in, "%d", &best[0]);
    sol = 1;
    for (i = 1; i < n; i++) {
        best[i] = -1;
          fscanf (in, "%d", &x);

        if(x >= best[0]) {
          k = 0;
          best[0] = x;
        }
        else {
          // use binary search instead
          low = 0;
          high = sol-1;
          for(;;) {
            k = (int) (low + high) / 2;
            // go lower in the array
            if(x > best[k] && x > best[k-1]) {
              high = k - 1;
              continue;
            }
            // go higher in the array
            if(x < best[k] && x < best[k+1]) {
              low = k + 1;
              continue;
            }
            // check if right spot
            if(x > best[k] && x < best[k-1])
              best[k] = x;
            if(x < best[k] && x > best[k+1])
              best[++k] = x;
            break;
          }
        }
              sol = MAX (sol, k + 1);
    }
    printf ("best is %d\n", sol);
    fclose(in);
    return 0;
}
```

## Summary

These programs demonstrate the main concept behind dynamic programming: build larger solutions based on previously found solutions. This building-up of solutions often yields programs that run very quickly.

For the previous programming challenge, the main subproblem was: Find the largest decreasing subsequence (and its first value) for numbers to the `right' of a given element.

Note that this sort of approach solves a class of problems that might be denoted ``one-dimensional''.

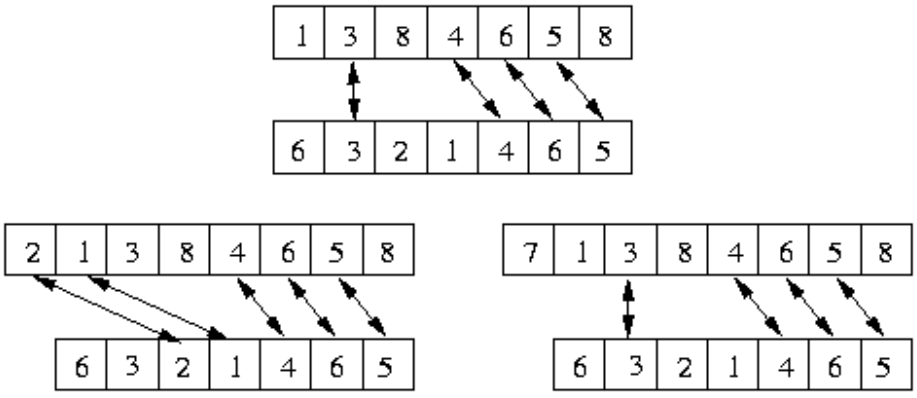
**Two Dimensional DP**

It is possible to create higher dimension problems such as:

Given two sequences of integers, what is the longest sequence which is a subsequence of both sequences?

Here, the subproblems are the longest common subsequence of smaller sequences (where the sequences are the tails of the original subsequences). First, if one of the sequences contains only one element, the solution is trivial (either the element is in the other sequence or it isn't).

Look at the problem of finding the longest common subsequence of the last i elements of the first sequence and the last j elements of the second sequences. There are only two possibilities. The first element of the first tail might be in the longest common subsequence or it might not. The longest common sequence not containing the first element of the first tail is merely the longest common subsequence of the last i-1 elements of the first sequence and the last j elements of the second subsequence. The other possibility results from some element in the tail of the second sequence matching the first element in tail of the first, and finding the longest common subsequence of the elements after those matched elements.




**Pseudocode**

Here's the pseudocode for this algorithm:

```
   # the tail of the second sequence is empty
1   for element = 1 to length1
2     length[element, length2+1] = 0


   # the tail of the first sequence has one element
3   matchelem = 0
```

```
  for element = length2 to 1
    if list1[length1] = list2[element]
      matchelem = 1
    length[length1,element] = nmatchlen

  # loop over the beginning of the tail of the first sequence
  for loc = length1-1 to 1
    maxlen = 0
    for element = length2 to 1
  # longest common subsequence doesn't include first element
      if length[loc+1,element] > maxlen
        maxlen = length[loc+1,element]
  # longest common subsequence includes first element
      if list1[loc] = list2[element] &&
                      length[loc+1,element+1]+1 > maxlen
          maxlen = length[loc,element+1] + 1
      length[loc,element] = maxlen
```

This program runs in O(*N* x *M*) time, where *N* and *M* are the respectively lengths of the sequences.

Note that this algorithm does not directly calculate the longest common subsequence. However, given the length matrix, you can determine the subsequence fairly quickly:

```
  location1 = 1
  location2 = 1


  while (length[location1,location2] != 0)
    flag = False
    for element = location2 to length2
      if (list1[location1] = list2[element] AND
         length[location1+1,element+1]+1 = length[location1,location2]
        output (list1[location1],list2[element])
        location2 = element + 1
        flag = True
        break for
      location1 = location1 + 1
```

The trick to dynamic programming is finding the subproblems to solve. Sometimes it involves multiple parameters:

A bitonic sequence is a sequence which increases for the first part and decreases for the second part. Find the longest bitonic sequence of a sequence of integers (technically, a bitonic can either increase-then-decrease or decrease-then-increase, but for this problem, only increase and then decrease will be considered).

In this case, the subproblems are the longest bitonic sequence and the longest decreasing sequence of prefixes of the sequence (basically, what's longest sequence assuming the turn has not occurred yet, and what's longest sequence starting here assuming the turn has already occurred).

Sometimes the subproblems are well hidden:

You have just won a contest where the prize is a free vacation in Canada. You must travel via air, and the cities are ordered from east to west. In addition, according to the rules, you must start at the further city west, travel only east until you reach the furthest city east, and then fly only west until you reach your starting location. In addition, you may visit no city more than once (except the starting city, of course).

Given the order of the cities, with the flights that can be done (you can only fly between certain cities, and just because you can fly from city A to city B does not mean you can fly the other direction), calculate the maximum number of cities you can visit.

The obvious item to try to do dynamic programming on is your location and your direction, but it's important what path you've taken (since you can't revisit cities on the

return trip), and the number of paths is too large to be able to solve (and store) the result of doing all of those subproblems.

However, if, instead of trying to find the path as described, it is found a different manner, then the number of states greatly decreases. Imagine having two travelers who start in the western most city. The travelers take turns traveling east, where the next traveler to move is always the western-most, but the travelers may never be at the same city, unless it is either the first or the last city. However, one of the traveler is only allowed to make "reverse flights," where he can travel from city A to city B if and only if there is a flight from city **B** to city **A**.

It's not too difficult to see that the paths of the two travelers can be combined to create a round-trip, by taking the normal traveler's path to the eastern-most city, and then taking the reverse of the other traveler's path back to the western-most city. Also, when traveler x is moved, you know that the traveler y has not yet visited any city east of traveler x except the city traveler y is current at, as otherwise traveler y must have moved once while x was west of y. Thus, the two traveler's paths are disjoint. Why this algorithm might yield the maximum number of cities is left as an exercise.

### Recognizing Problems solvable by dynamic programming

Generally, dynamic programming solutions are applied to those solutions which would otherwise be exponential in time, so if the bounds of the problem are too large to be able to be done in exponential time for any but a very small set of the input cases, look for a dynamic programming solution. Basically, any program you were thinking about doing recursive descent on, you should check to see if a dynamic programming solution exists if the inputs limits are over 30.

### Finding the Subproblems

As mentioned before, finding the subproblems to do dynamic programming over is the key. Your goal is to completely describe the state of a solution in a small amount of data, such as an integer, a pair of integers, a boolean and an integer, etc.

Almost without fail, the subproblem will be the `tail-end' of a problem. That is, there is a way to do the recursive descent such that at each step, you only pass a small amount of data. For example, in the air travel one, you could do recursive descent to find the complete path, but that means you'd have to pass not only your location, but the cities you've visited already (either as a list or as a boolean array). That's too much state for dynamic programming to work on. However, recursing on the pair of cities as you travel east subject to the constant given is a very small amount of data to recurse on.

If the path is important, you will not be able to do dynamic programming unless the paths are **very** short. However, as in the air travel problem, depending on how you look at it, the path may not be important.

### Sample Problems

Polygon Game [1998 IOI]

Imagine a regular N-gon. Put numbers on nodes, either and the operators `+' or `*' on the edges. The first move is to remove an edge. After that, combine (e.g., evaluate the simple term) across edges, replacing the edge and end points with node with value

equal to value of end point combined by operations, for example:

```
  ...-- 3 --+-- 5 --*-- 7 --...
  ...----- 8 ----*----- 7 --...
  ...--------- 56 -----------...
```

Given a labelled N-gon, maximize the final value computed.

**Subset Sums [Spring 98 USACO]**

For many sets of consecutive integers from 1 through N (1 <= N <= 39), one can partition the set into two sets whose sums are identical. For example, if N=3, one can partition the set {1, 2, 3} in one way so that the sums of both subsets are identical:

{3} and {1,2}

This counts as a single partitioning (i.e., reversing the order counts as the same partitioning and thus does not increase the count of partitions).

If N=7, there are 4 ways to partition the set {1, 2, 3, ... 7} so that each partition has the same sum:

{1,6,7} and {2,3,4,5}
{2,5,7} and {1,3,4,6}
{3,4,7} and {1,2,5,6}
{1,2,4,7} and {3,5,6}

Given N, your program should print the number of ways a set containing the integers from 1 through N can be partitioned into two sets whose sums are identical. Print 0 if there are no such ways.

**Number Game [IOI 96, maybe]**

Given a sequence of no more than 100 integers (-32000..32000), two opponents alternate turns removing the leftmost or rightmost number from a sequence. Each player's score at the end of the game is the sum of those numbers he or she removed. Given a sequence, determine the maximum winning score for the first player, assuming the second player plays optimally.

USACO Gateway | Comment or Question

# Eulerian Tour

### Sample Problem: Riding The Fences

Farmer John owns a large number of fences, which he must periodically check for integrity. Farmer John keeps track of his fences by maintaining a list of their intersection points, along with the fences which end at each point. Each fence has two end points, each at an intersection point, although the intersection point may be the end point of only a single fence. Of course, more than two fences might share an endpoint.

Given the fence layout, calculate if there is a way for Farmer John to ride his horse to all of his fences without riding along a fence more than once. Farmer John can start and end anywhere, but cannot cut across his fields (the only way he can travel between intersection points is along a fence). If there is a way, find one way.

### The Abstraction

Given: An undirected graph

Find a path which uses every edge exactly once. This is called an Eulerian tour. If the path begins and ends at the same vertex, it is called a Eulerian circuit.

### The Algorithm

Detecting whether a graph has an Eulerian tour or circuit is actually easy; two different rules apply.

- A graph has an Eulerian circuit if and only if it is connected (once you throw out all nodes of degree 0) and every node has `even degree'.
- A graph has an Eulerian path if and only if it is connected and every node except two has even degree.
- In the second case, one of the two nodes which has odd degree must be the start node, while the other is the end node.

The basic idea of the algorithm is to start at some node the graph and determine a circuit back to that same node. Now, as the circuit is added (in reverse order, as it turns out), the algorithm ensures that all the edges of all the nodes along that path have been used. If there is some node along that path which has an edge that has not been used, then the algorithm finds a circuit starting at that node which uses that edge and splices this new circuit into the current one. This continues until all the edges of every node in the original circuit have been used, which, since the graph is connected, implies that all the edges have been used, so the resulting circuit is Eulerian.

More formally, to determine a Eulerian circuit of a graph which has one, pick a starting node and recurse on it. At each recursive step:

- Pick a starting node and recurse on that node. At each step:

- If the node has no neighbors, then append the node to the circuit and return
- If the node has a neighbor, then make a list of the neighbors and process them (which includes deleting them from the list of nodes on which to work) until the node has no more neighbors
- To process a node, delete the edge between the current node and its neighbor, recurse on the neighbor, and postpend the current node to the circuit.

And here's the pseudocode:

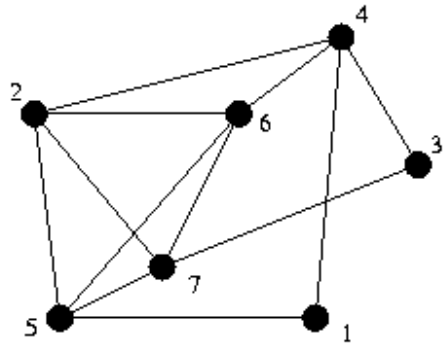
```
# circuit is a global array
   find_euler_circuit
     circuitpos = 0
     find_circuit(node 1)

# nextnode and visited is a local array
# the path will be found in reverse order
  find_circuit(node i)

    if node i has no neighbors then
      circuit(circuitpos) = node i
      circuitpos = circuitpos + 1
    else
      while (node i has neighbors)
          pick a random neighbor node j of node i
          delete_edges (node j, node i)
          find_circuit (node j)
      circuit(circuitpos) = node i
      circuitpos = circuitpos + 1
```

To find an Eulerian tour, simply find one of the nodes which has odd degree and call `find_circuit` with it.

Both of these algorithms run in O(m + n) time, where m is the number of edges and n is the number of nodes, if you store the graph in adjacency list form. With larger graphs, there's a danger of overflowing the run-time stack, so you might have to use your own stack.

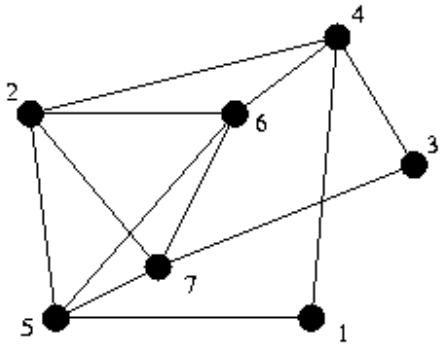
**Execution Example**

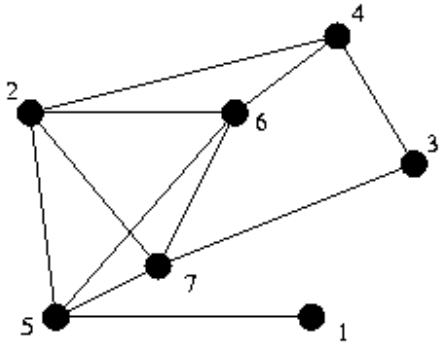
Consider the following graph:



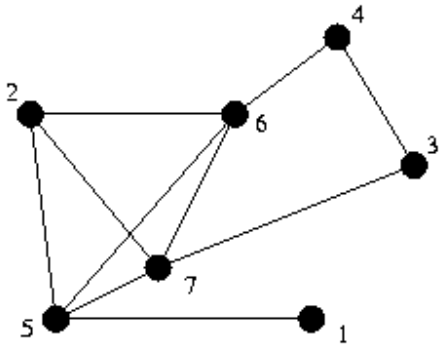

Assume that selecting a random neighbor yields the lowest numbered neighbor, the execution goes as follows:

Stack:
Location: 1
Circuit:

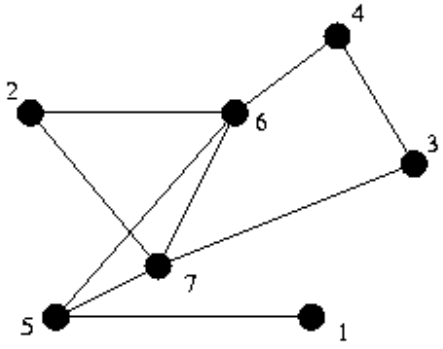



Stack: 1
Location: 4
Circuit:

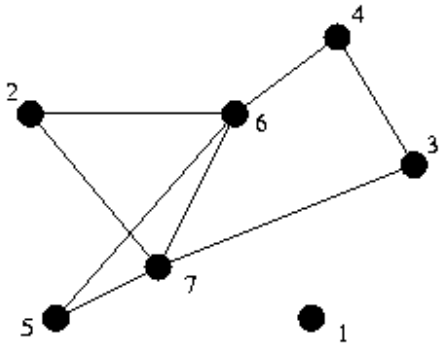


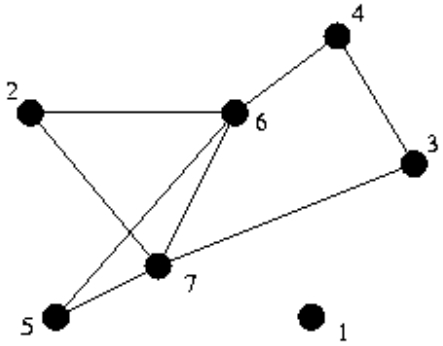

Stack: 1 4
Location: 2
Circuit:



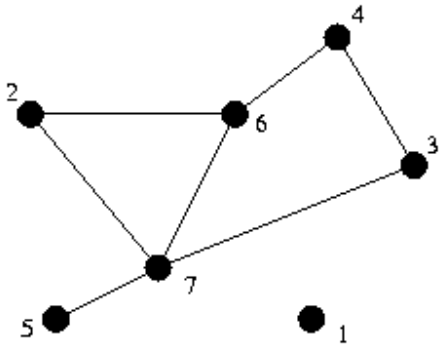

Stack: 1 4 2
Location: 5
Circuit:

Stack: 1 4 2 5
Location: 1
Circuit:

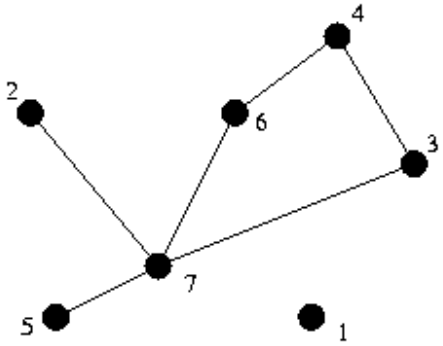



Stack: 1 4 2
Location: 5
Circuit: 1

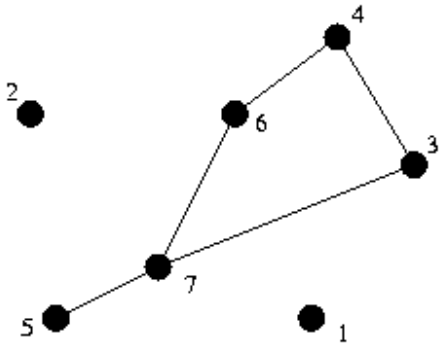


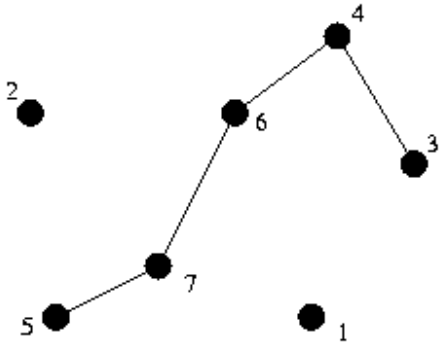

Stack: 1 4 2 5
Location: 6
Circuit: 1



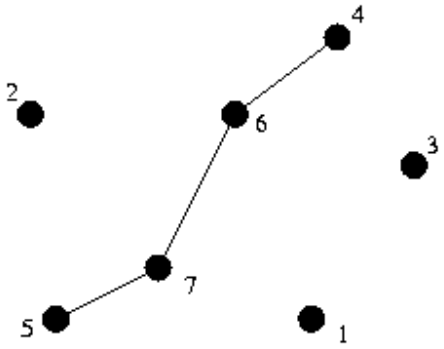

Stack: 1 4 2 5 6
Location: 2
Circuit: 1

Stack: 1 4 2 5 6 2
Location: 7
Circuit: 1

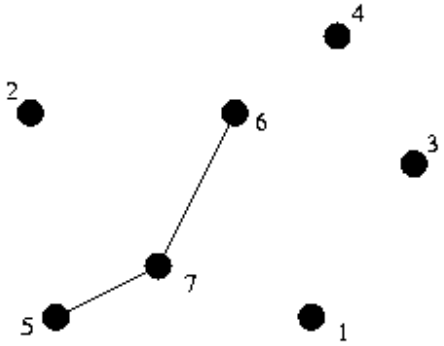



Stack: 1 4 2 5 6 2 7
Location: 3
Circuit: 1

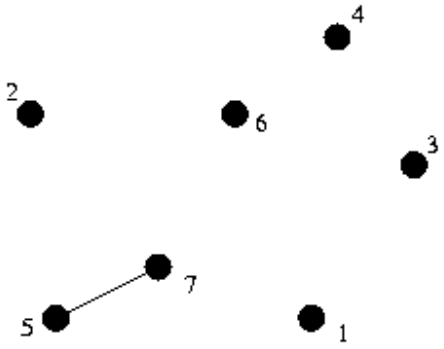


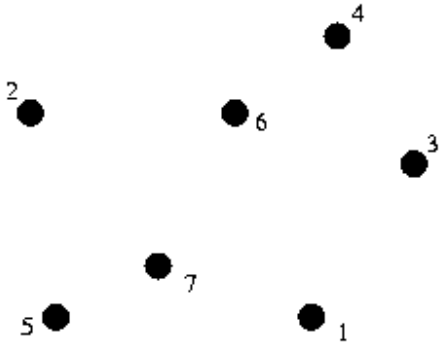

Stack: 1 4 2 5 6 2 7 3
Location: 4
Circuit: 1



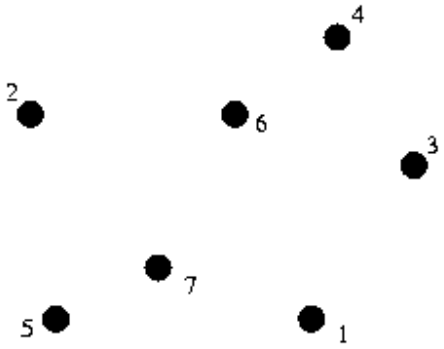

Stack: 1 4 2 5 6 2 7 3 4
Location: 6
Circuit: 1

Stack: 1 4 2 5 6 2 7 3 4 6
Location: 7
Circuit: 1

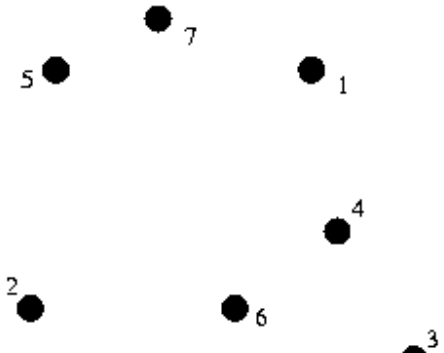


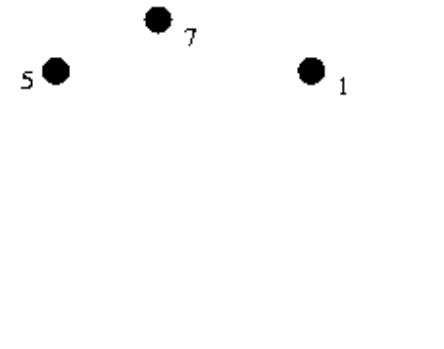

Stack: 1 4 2 5 6 2 7 3 4 6 7
Location: 5
Circuit: 1



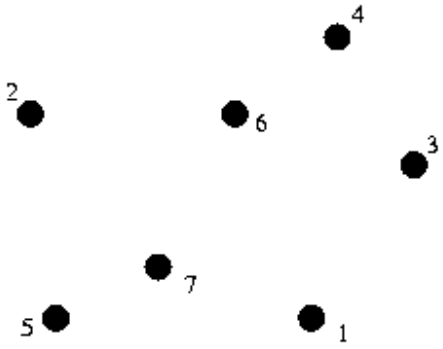

Stack: 1 4 2 5 6 2 7 3 4 6
Location: 7
Circuit: 1 5

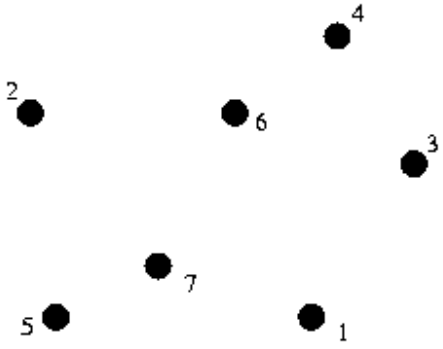



Stack: 1 4 2 5 6 2 7 3 4
Location: 6
Circuit: 1 5 7

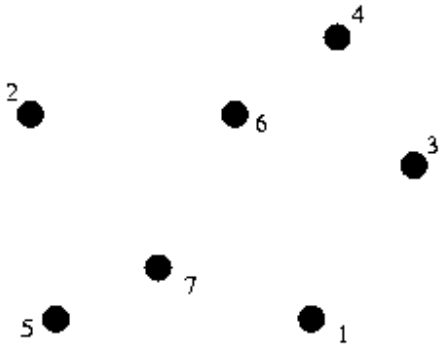



Stack: 1 4 2 5 6 2 7 3
Location: 4
Circuit: 1 5 7 6

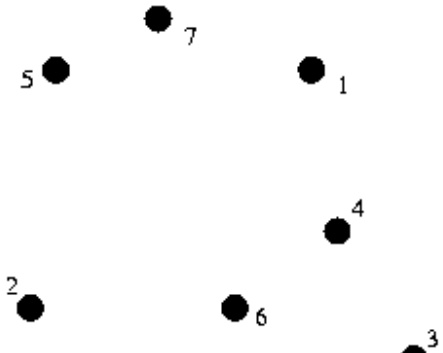


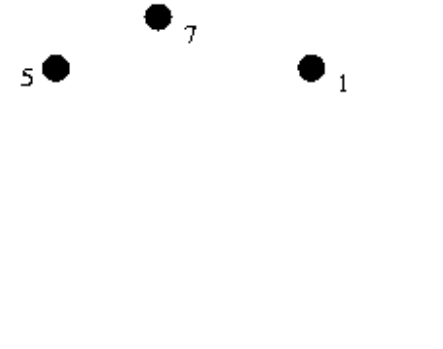

Stack: 1 4 2 5 6 2 7
Location: 3
Circuit: 1 5 7 6 4



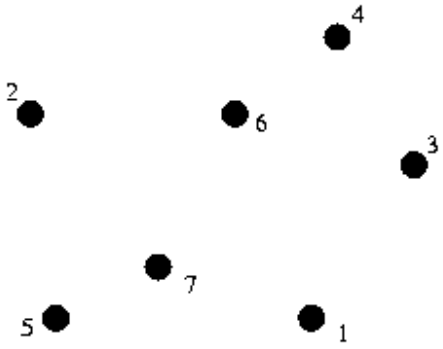

Stack: 1 4 2 5 6 2
Location: 7
Circuit: 1 5 7 6 4 3

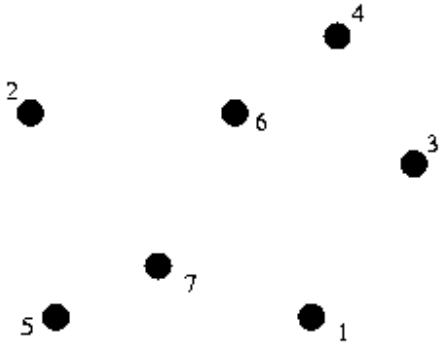



Stack: 1 4 2 5 6
Location: 2
Circuit: 1 5 7 6 4 3 7

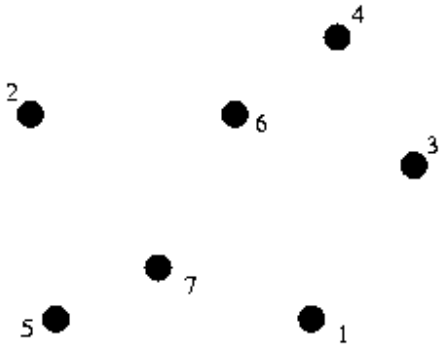



Stack: 1 4 2 5
Location: 6
Circuit: 1 5 7 6 4 3 7 2

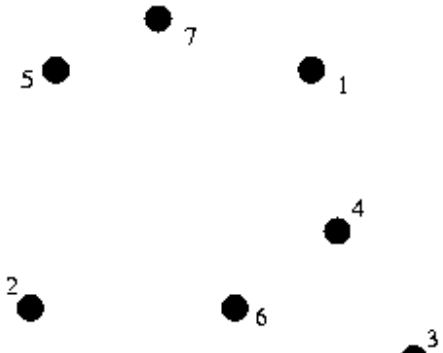


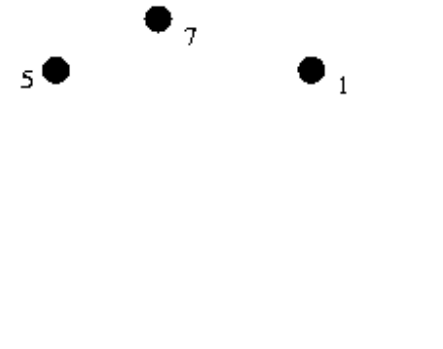

Stack: 1 4 2
Location: 5
Circuit: 1 5 7 6 4 3 7 2 6



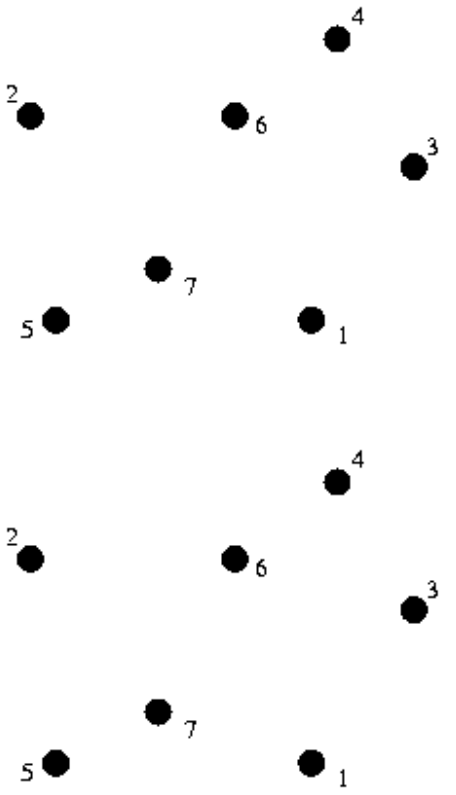

Stack: 1 4
Location: 2
Circuit: 1 5 7 6 4 3 7 2 6 5




Stack: 1
Location: 4
Circuit: 1 5 7 6 4 3 7 2 6 5 2




Stack:
Location: 1
Circuit: 1 5 7 6 4 3 7 2 6 5 2 4




Stack:
Location:
Circuit: 1 5 7 6 4 3 7 2 6 5 2 4 1

## Extensions

Multiple edges between nodes can be handled by the exact same algorithm.

Self-loops can be handled by the exact same algorithm as well, if self-loops are considered to add 2 (one in and one out) to the degree of a node.

A directed graph has a Eulerian circuit if it is strongly connected (except for nodes with both in-degree and out-degree of 0) and the indegree of each node equals its outdegree. The algorithm is exactly the same, except that because of the way this code finds the cycle, you must traverse arcs in reverse order.

Finding a Eulerian path in a directed graph is harder. Consult Sedgewick if you are interested.

## Example problems

#### Airplane Hopping

Given a collection of cities, along with the flights between those cities, determine if there is a sequence of flights such that you take every flight exactly once, and end up at the place you started.

Analysis: This is equivalent to finding a Eulerian circuit in a directed graph.

#### Cows on Parade

Farmer John has two types of cows: black Angus and white Jerseys. While marching 19 of their cows to market the other day, John's wife Farmeress Joanne, noticed that all 16

possibilities of four successive black and white cows (e.g., bbbb, bbbw, bbwb, bbww, ..., wwww) were present. Of course, some of the combinations overlapped others.

Given N (2 <= N <= 15), find the minimum length sequence of cows such that every combination of N successive black and white cows occurs in that sequence.

Analysis: The vertices of the graph are the possibilities of N-1 cows. Being at a node corresponds to the last N-1 cows matching the node in color. That is, for N = 4, if the last 3 cows were *wbw*, then you are at the *wbw* node. Each node has out-degree of 2, corresponding to adding a black or white cow to the end of the sequence. In addition, each node has in-degree of 2, corresponding to whether the cow just before the last N-1 cows is black or white.

The graph is strongly connected, and the in-degree of each node equals its out-degree, so the graph has a Eulerian circuit.

The sequence corresponding to the Eulerian circuit is the sequence of N-1 cows of the first node in the circuit, followed by cows corresponding to the color of the edge.

USACO Gateway | Comment or Question

# Flood Fill

### Sample Problem: Connected Fields

Farmer John's fields are broken into fields, with paths between some of them. Unfortunately, some fields are not reachable from other fields via the paths.

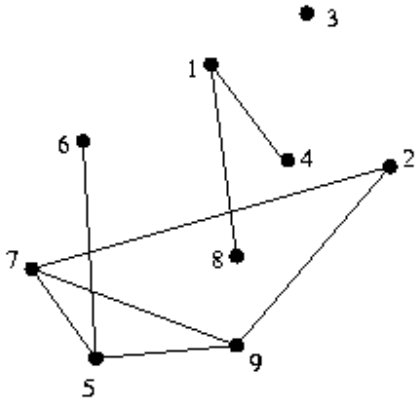
Define a *superfield* is a collection of fields that are all reachable from each other. Calculate the number of superfields.

### The Abstraction

Given: a undirected graph

The *component* of a graph is a maximal-sized (though not necessarily maximum) subgraph which is connected.

Calculate the component of the graph.




This graph has three components: {1,4,8}, {2,5,6,7,9}, and {3}.

### The Algorithm: Flood Fill

Flood fill can be performed three basic ways: depth-first, breadth-first, and breadth-first scanning. The basic idea is to find some node which has not been assigned to a component and to calculate the component which contains. The question is how to calculate the component.

In the depth-first formulation, the algorithm looks at each step through all of the neighbors of the current node, and, for those that have not been assigned to a component yet, assigns them to this component and recurses on them.

In the breadth-first formulation, instead of recursing on the newly assigned nodes, they are added to a queue.

In the breadth-first scanning formulation, every node has two values: component and visited. When calculating the component, the algorithm goes through all of the nodes that have been assigned to that component but not visited yet, and assigns their neighbors to the current component.

The depth-first formulation is the easiest to code and debug, but can require a stack as big as the original graph. For explicit graphs, this is not so bad, but for implicit graphs, such as the problem presented has, the numbers of nodes can be very large.

The breadth-formulation does a little better, as the queue is much more efficient than the run-time stack is, but can still run into the same problem. Both the depth-first and breadth-first formulations run in N + M time, where N is the number of vertices and M is the number of edges.

The breadth-first scanning formulation, however, requires very little extra space. In fact, being a little tricky, it requires no extra space. However, it is slower, requiring up to N*N + M time, where N is the number of vertices in the graph.

### Pseudocode for Breadth-First Scanning

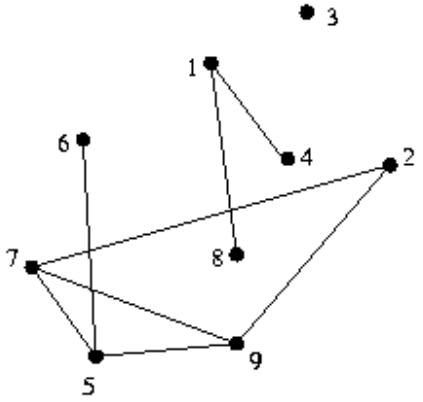
This code uses a trick to not use extra space, marking nodes to be visited as in component -2 and actually assigning them to the current component when they are actually visited.

```
# component(i) denotes the
# component that node i is in
 1 function flood_fill(new_component)

 2 do
 3   num_visited = 0
 4   for all nodes i
 5     if component(i) = -2
 6       num_visited = num_visited + 1
 7       component(i) = new_component
 8       for all neighbors j of node i
 9         if component(j) = nil
10           component(j) = -2
11 until num_visited = 0

12 function find_components

13  num_components = 0
14  for all nodes i
15    component(node i) = nil
16  for all nodes i
17    if component(node i) is nil
18      num_components =
                 num_components + 1
19      component(i) = -2
20      flood_fill(component
                       num_components)
```

Running time of this algorithm is $O(N^2)$, where $N$ is the numbers of nodes. Every edge is traversed twice (once for each end-point), and each node is only marked once.

### Execution Example

Consider the graph from above.




The algorithm starts with all nodes assigned to no component.

Going through the nodes in order first node not assigned to any component yet is vertex 1. Start a new component (component 1) for that node, and set the component of node 1 to -2 (any nodes not shown are unassigned).

| Node | Component |
|---|---|
| **1** | **-2** |

Now, in the `flood_fill` code, the first time through the `do` loop, it finds the node 1 is assigned to component -2. Thus, it reassigns it to component 1, signifying that it has been visited, and then assigns its neighbors (node 4) to component -2.

| Node | Component |
|---|---|
| **1** | **1** |
| **4** | **-2** |

As the loop through all the nodes continues, it finds that node 4 is also assigned to component -2, and processes it appropriately as well.

| Node | Component |
|---|---|
| 1 | 1 |
| **4** | **1** |
| **8** | **-2** |

Node 8 is the next to be processed.

| Node | Component |
|---|---|
| 1 | 1 |
| 4 | 1 |
| **8** | **1** |

Now, the `for` loop continues, and finds no more nodes that have not been assigned yet. Since the `until` clause is not satisfied ( `num_visited` = 3), it tries again. This time, no nodes are found, so the function exits and component 1 is complete.

The search for unassigned nodes continues, finding node 2. A new component (component 2) is allocated, node 2 is marked as in component -2, and `flood_fill` is called.

| Node | Component |
|---|---|
| 1 | 1 |
| **2** | **-2** |
| 4 | 1 |
| 8 | 1 |

Node 2 is found as marked in component -2, and is processed.

| Node | Component |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 4 | 1 |
| **7** | **-2** |
| 8 | 1 |
| **9** | **-2** |

Next, node 7 is processed.

| Node | Component |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 4 | 1 |
| **5** | **-2** |
| **7** | **2** |
| 8 | 1 |
| 9 | -2 |

Then node 9 is processed.

| Node | Component |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 4 | 1 |
| 5 | -2 |
| 7 | 2 |
| 8 | 1 |
| **9** | **2** |

The terminating condition does not hold ( `num_visited` = 3), so the search through for nodes assigned to component -2 starts again. Node 5 is the first one found.

| Node | Component |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 4 | 1 |
| 5 | 2 |
| **6** | **-2** |
| 7 | 2 |
| 8 | 1 |
| 9 | 2 |

Node 6 is the next node found to be in component -2.

| Node | Component |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 4 | 1 |
| 5 | 2 |
| **6** | **2** |
| 7 | 2 |
| 8 | 1 |
| 9 | 2 |

No more nodes are found assigned to component -2, but the terminating condition does not hold, so one more pass through the nodes is performed, finding no nodes assigned to component -2. Thus, the search for unassigned nodes continue from node 2, finding node 3 unassigned.

| Node | Component |
|---|---|
| 1 | 1 |
| 2 | 2 |
| **3** | **-2** |
| 4 | 1 |
| 5 | 2 |
| 6 | 2 |
| 7 | 2 |
| 8 | 1 |
| 9 | 2 |

Node 3 is processed.

| Node | Component |
|---|---|
| 1 | 1 |
| 2 | 2 |

| | |
|---|---|
| 3 | 3 |
| 4 | 1 |
| 5 | 2 |
| 6 | 2 |
| 7 | 2 |
| 8 | 1 |
| 9 | 2 |

From here, the algorithm eventually terminates, as there are no more nodes assigned to component -2 and no unassigned nodes. The three components of the graph have been determined, along with the component to which each node belongs.

### Problem Cues

Generally, these types of problem are fairly clear. If it asks for sets of "connected" things, it's probably asking for components, in which case flood fill works very well. Often, this is a step in solving the complete problem.

### Extensions

The notion of ``components'' becomes muddied when you go to directed graphs.

However, the same flooding idea can be used to determine the points which are reachable from any given point even in a directed graph. At each recursive step, if the point isn't marked already, mark the point as reachable and recurse on all of its neighbors.

Note that to determine which points can reach a given point in a directed graph can be solved the same, by looking at every arc backwards.

### Sample Problems

#### Company Ownership [abridged, IOI 93]

Given: A weighted directed graph, with weights between 0 and 100.

Some vertex A ``owns'' another vertex B if:

- A = B
- There is an arc from A to B with weight more than 50.
- There exists some set of vertices $C_1$ through $C_k$ such that A owns $C_1$ through $C_k$, and each vertex has an arc of weight $x_1$ through $x_k$ to vertex B, and $x_1 + x_2 + ... + x_k > 50$.

Find all (a,b) pairs such that a owns b.

Analysis: This can be solved via an adaptation of the calculating the vertices reachable from a vertex in a directed graph. To calculate which vertices vertex A owns, keep track of the ``ownership percentage'' for each node. Initialize them all to zero. Now, at each recursive step, mark the node as owned by vertex A and add the weight of all outgoing

arcs to the ``ownership percentages.'' For all percentages that go above 50, recurse into those vertices.

**Street Race [IOI 95]**

Given: a directed graph, and a start point and an end point.

Find all points p that any path from the start point to the end must travel through p.

Analysis: The easiest algorithm is to remove each point in turn, and check to see if the end point is reachable from the start point. This runs in O(N (M + N)) time. Since the original problem stated that M <= 100, and N <= 50, this will run in time easily.

**Cow Tours [1999 USACO National Championship, abridged]**

The diameter of a connected graph is defined as the maximum distance between any two nodes of the graph, where the distance between two nodes is defined as the length of the shortest path.

Given a set of points in the plane, and the connections between those points, find the two points which are currently not in the same component, such that the diameter of the resulting component is minimized.

Analysis: Find the components of the original graph, using the method described above. Then, for each pair of points not in the same component, try placing a connection between them. Find the pair that minimizes the diameter.

**Connected Fields**

Farmer John contracted out the building of a new barn. Unfortunately, the builder mixed up the plans of Farmer John's barn with another set of plans. Farmer John's plans called for a barn that only had one room, but the building he got might have many rooms. Given a grid of the layout of the barn, tell Farmer John how many rooms it has.

Analysis: The graph here is on the non-wall grid locations, with edge between adjacent non-wall locations, although the graph should be stored as the grid, and not transformed into some other form, as the grid is so compact and easy to work with.

USACO Gateway | Comment or Question

# Graph Theory

## What's a Graph?

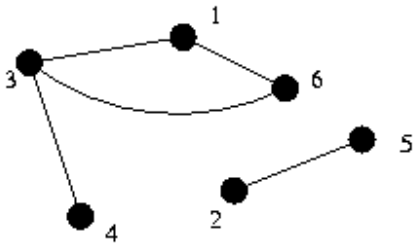
Formally, a *graph* is the following:

- a collection of *vertices* V, and
- a collection of *edges* E consisting of pairs of vertices.

Think of vertices as ``locations''. The set of vertices is the set of all the possible locations. In this analogy, edges represent paths between pairs of those locations; the set E contains all the paths between the locations.

## Representation

The graph is normally represented using that analogy. Vertices are points or circles; edges are lines between them.




In this example graph, V = {1, 2, 3, 4, 5, 6} and E = {(1,3), (1,6), (2,5), (3,4), (3,6)}.

Each *vertex* is a member of the set V. A vertex is sometimes called a *node*.

Each *edge* is a member of the set E. Note that some vertices might not be the end point of any edge. Such vertices are termed `isolated'.

Sometimes, numerical values are associated with edges, specifying lengths or costs; such graphs are called *edge-weighted* graphs (or weighted graphs). The value associated with an edge is called the *weight* of the edge. A similar definition holds for node-weighted graphs,

## Examples of Graphs

### Telecowmunication (USACO Championship 1996)

Given a set of computers and a set of wires running between pairs of computers, what is the minimum number of machines whose crash causes two given machines to be unable to communicate? (The two given machines will not crash.)

Graph: The vertices of the graph are the computers. The edges are the wires between the computers.

#### Sample Problem: Riding The Fences

Farmer John owns a large number of fences, which he must periodically check for integrity. He keeps track of his fences by maintaining a list of points at which fences intersect. He records the name of the point and the one or two fence names that touch that point. Every fence has two end points, each at some intersection point, although the intersection point may be the end point of only one fence.

Given a fence layout, calculate if there is a way for Farmer John to ride his horse to all of his fences without riding along a fence more than once. Farmer John can start and finish anywhere, but cannot cut across his fields (i.e., the only way he can travel between intersection points is along a fence). If there is a way, find one way.

Graph: Farmer John starts at intersection points and travels between the points along fences. Thus, the vertices of the underlying graph are the intersection points, and the fences represent edges.

#### Knight moves

Given: Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.

Graph: The graph here is harder to see. Each location on the chessboard represents a vertex. There is an edge between two positions if it is a legal knight move.

#### Overfencing [Kolstad & Schrijvers, Spring 1999 USACO Open]

Farmer John created a huge maze of fences in a field. He omitted two fence segments on the edges, thus creating two ``exits'' for the maze. The maze is a `perfect' maze; you can find a way out of the maze from any point inside it.

Given the layout of the maze, calculate the number of steps required to exit the maze from the `worst' point in the maze (the point that is `farther' from either exit when walking optimally to the closest exit).

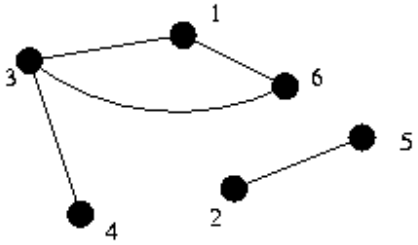
Here's what one particular W=5, H=3 maze looks like:

```
+-+-+-+-+-+
|         |
+-+ +-+ + +
|     | | |
+ +-+-+ + +
| |     |
+-+ +-+-+-+
```

Graph: The vertices of the graph are positions in the grid. There is an edge between two vertices if they represent adjacent positions that are not separated by a wall.

## Terminology

Let's look again at the first example graph:




An edge is a *self-loop* if it is of the form (u,u). The sample graph contains no self-loops.

A graph is *simple* if it neither contains self-loops nor contains an edge that is repeated in E. A graph is called a *multigraph* if it contains a given edge more than once or contain self-loops. For our discussions, graphs are assumed to be simple. The example graph is a simple graph.

An edge (u,v) is *incident* to both vertex u and vertex v. For example, the edge (1,3) is incident to vertex 3.

The *degree* of a vertex is the number of edges which are incident to it. For example, vertex 3 has degree 3, while vertex 4 has degree 1.

Vertex u is *adjacent* to vertex v if there is some edge to which both are incident (that is, there is an edge between them). For example, vertex 2 is adjacent to vertex 5.

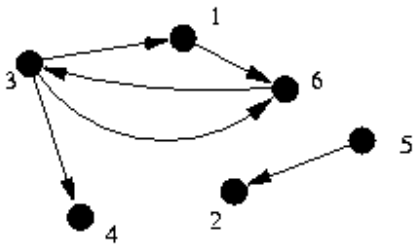
A graph is said to be *sparse* if the total number of edges is small compared to the total number possible ((*N x (N-1))/2*) and *dense* otherwise. For a given graph, whether it is dense or sparse is not well-defined.

**Directed Graph**

Graphs described thus far are called *undirected*, as the edges go `both ways'. So far, the graphs have connoted that if one can travel from vertex 1 to vertex 3, one can also travel from vertex 3 to vertex 1. In other words, (1,3) being in the edge set implies (3,1) is in the edge set.

Sometimes, however, a graph is *directed*, in which case the edges have a direction. In this case, the edges are called *arcs*.

Directed graphs are drawn with arrows to show direction.



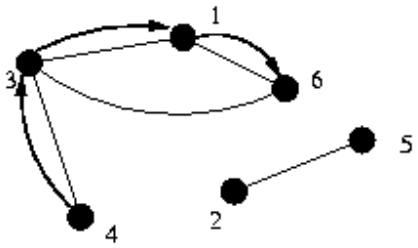

The *out-degree* of a vertex is the number of arcs which *begin* at that vertex. The *in-degree* of a vertex is the number of arcs which *end* at that vertex. For example, vertex 6 has in-degree 2 and out-degree 1.

A graph is assumed to be undirected unless specifically called a directed graph.

### Paths

A *path* from vertex $u$ to vertex $x$ is a sequence of vertices ($v_0$, $v_1$, ..., $v_k$) such that $v_0 = u$ and $v_k = x$ and ($v_0$, $v_1$) is an edge in the graph, as is ($v_1$, $v_2$), ($v_2$, $v_3$), etc. The length of such a path is $k$.

For example, in the undirected graph above, (4, 3, 1, 6) is a path.




This path is said to *contain* the vertices $v_0$, $v_1$, etc., as well as the edges ($v_0$, $v_1$), ($v_1$, $v_2$), etc.

Vertex $x$ is said to be *reachable* from vertex $u$ if a path exists from $u$ to $x$.

A path is *simple* if it contains no vertex more than once.

A path is a *cycle* if it is a path from some vertex to that same vertex. A cycle is *simple* if it contains no vertex more than once, except the start (and end) vertex, which only appears as the first and last vertex in the path.

These definitions extend similarly to directed graphs (e.g., ($v_0$, $v_1$), ($v_1$, $v_2$), etc. must be arcs).

### Graph Representation

The choice of representation of a graph is important, as different representations have very different time and space costs.

The vertices are generally tracked by numbering them, so that one can index them just by their number. Thus, the representations focus on how to store the *edges*.

#### Edge List

The most obvious way to keep track of the edges is to keep a list of the pairs of vertices representing the edges in the graph.

This representation is easy to code, fairly easy to debug, and fairly space efficient. However, determining the edges incident to a given vertex is expensive, as is determining if two vertices are adjacent. Adding an edge is quick, but deleting one is difficult if its location in the list is not known.

For weighted graphs, this representation also keeps one more number for each edge, the edge weight. Extending this data structure to handle directed graphs is

straightforward. Representing multigraphs is also trivial.

**Example**

The sample undirected graph might be represented as the following list of edges:

| | $V_1$ | $V_2$ |
|---|---|---|
| $e_1$ | 4 | 3 |
| $e_2$ | 1 | 3 |
| $e_3$ | 2 | 5 |
| $e_4$ | 6 | 1 |
| $e_5$ | 3 | 6 |

**Adjacency Matrix**

A second way to represent a graph utilizes an *adjacency matrix*. This is a N by N array (N is the number of vertices). The i,j entry contains a 1 if the edge (i,j) is in the graph; otherwise it contains a 0. For an undirected graph, this matrix is symmetric.

This representation is easy to code. It's much less space efficient, especially for large, sparse graphs. Debugging is harder, as the matrix is large. Finding all the edges incident to a given vertex is fairly expensive (linear in the number of vertices), but checking if two vertices are adjacent is very quick. Adding and removing edges are also very inexpensive operations.

For weighted graphs, the value of the (i,j) entry is used to store the weight of the edge. For an unweighted multigraph, the (i,j) entry can maintain the number of edges between the vertices. For a weighted multigraph, it's harder to extend this.

**Example**

The sample undirected graph would be represented by the following adjacency matrix:

| | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|---|---|---|---|---|---|---|
| $V_1$ | 0 | 0 | 1 | 0 | 0 | 1 |
| $V_2$ | 0 | 0 | 0 | 0 | 1 | 0 |
| $V_3$ | 1 | 0 | 0 | 1 | 0 | 1 |
| $V_4$ | 0 | 0 | 1 | 0 | 0 | 0 |
| $V_5$ | 0 | 1 | 0 | 0 | 0 | 0 |
| $V_6$ | 1 | 0 | 1 | 0 | 0 | 0 |

It is sometimes helpful to use the fact that the (i,j) entry of the adjacency matrix raised to the k-th power gives the number of paths from vertex i to vertex j consisting of exactly k edges.

**Adjacency List**

The third representation of graph is to keep track of all the edges incident to a given vertex. This can be done by using an array of length N, where N is the number of vertices. The $i^{th}$ entry in this array is a list of the edges incident to i'th vertex (edges are represented by the index of the other vertex incident to that edge).

This representation is much more difficult to code, especially if the number of edges incident to each vertex is not bounded, so the lists must be linked lists (or dynamically allocated). Debugging this is difficult, as following linked lists is more difficult. However, this representation uses about as much memory as the edge list. Finding the vertices adjacent to each node is very cheap in this structure, but checking if two vertices are adjacent requires checking all the edges adjacent to one of the vertices. Adding an edge is easy, but deleting an edge is difficult, if the locations of the edge in the appropriate lists are not known.

Extend this representation to handle weighted graphs by maintaining both the weight and the other incident vertex for each edge instead of just the other incident vertex. Multigraphs are already representable. Directed graphs are also easily handled by this representation, in one of several ways: store only the edges in one direction, keep a seperate list of incoming and outgoing arcs, or denote the direction of each arc in the list.

**Example**

The adjacency list representation of the example undirected graph is as follows:

| | Adjacent |
|---|---|
| Vertex | Vertices |
| 1 | 3, 6 |
| 2 | 5 |
| 3 | 6, 4, 1 |
| 4 | 3 |
| 5 | 2 |
| 6 | 3, 1 |

**Implicit Representation**

For some graphs, the graph itself does not have to be stored at all. For example, for the Knight moves and Overfencing problems, it is easy to calculate the neighbors of a vertex, check adjacency, and determine all the edges without actually storing that information, thus, there is no reason to actually store that information; the graph is implicit in the data itself.

If it is possible to store the graph in this format, it is generally the correct thing to do, as it saves a lot on storage and reduces the complexity of your code, making it easy to both write and debug.

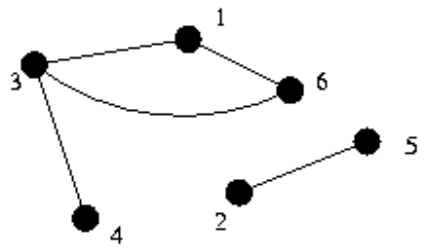
If N is the number of vertices, M the number of edges, and $d_{max}$ the maximum degree of a node, the following table summarizes the differences between the representations:

| Efficiency | Edge List | Adj Matrix | Adj List |
|---|---|---|---|
| | | | |

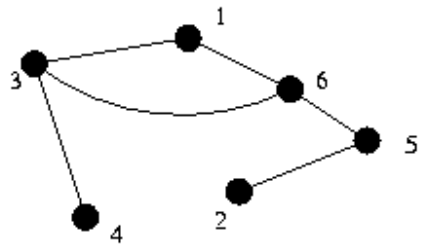
| Space | 2xM | $N^2$ | 2xM |
|---|---|---|---|
| Adjacency Check | M | 1 | $d_{max}$ |
| List of Adj Vertices | M | N | $d_{max}$ |
| Add Edge | 1 | 1 | 1 |
| Delete Edge | M | 2 | $2xd_{max}$ |

**Connectedness**

An undirected graph is said to be *connected* if there is a path from every vertex to every other vertex. The example graph is *not* connected, as there is no path from vertex 2 to vertex 4.




However, if you add an edge between vertex 5 and vertex 6, then the graph becomes connected.




A *component* of a graph is a maximal subset of the vertices such that every vertex is reachable from each other vertex in the component. The original example graph has two components: {1, 3, 4, 6} and {2, 5}. Note that {1, 3, 4} is not a component, as it is not maximal.

A directed graph is said to be *strongly connected* if there is a path from every vertex to every other vertex.

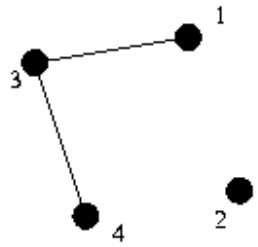
A *strongly connected component* of a directed graph is a vertex u and the collection of all vertices v such that there is a path from u to v and a path from v to u.

**Subgraphs**

Graph G' = (V', E') is a subgraph of G = (V, E) if V' is a subset of V and E' is a subset of E.

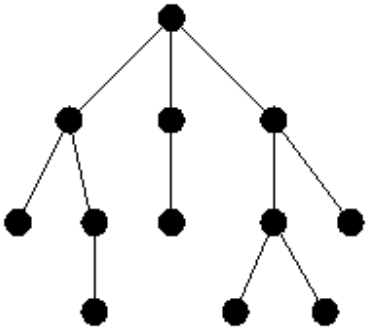
The subgraph of G *induced* by V' is the graph (V', E'), where E' consists of all the edges of E that are between members of V'.

For example, for V' = {1, 3, 4, 2}, the subgraph induced is:



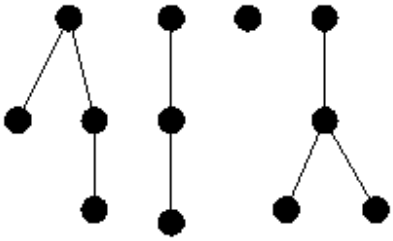

**Special Graphs**

An undirected graph is said to be a *tree* if it contains no cycles and is connected.



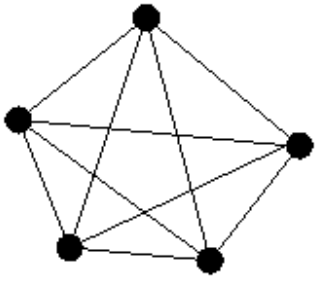
Many trees are what is called *rooted*, where there is a notion of the "top" node, which is called the *root*. Thus, each node has one *parent*, which is the adjacent node which is closer to the root, and may have any number of *children*, which are the rest of the nodes adjacent to it. The tree above was drawn as a rooted tree.

An undirected graph which contains no cycles is called a *forest*.

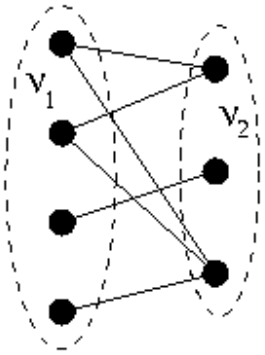


A directed acyclic graph is often referred to as a *dag*.

A graph is said to be *complete* if there is an edge between every pair of vertices.



A graph is said to be *bipartite* if the vertices can be split into two sets $V_1$ and $V_2$ such there are no edges between two vertices of $V_1$ or two vertices of $V_2$.




USACO Gateway | Comment or Question

# Greedy Algorithm

# Greedy Algorithm

**Sample Problem: Barn Repair [1999 USACO Spring Open]**

There is a long list of stalls, some of which need to be covered with boards. You can use up to N (1 <= N <= 50) boards, each of which may cover any number of consecutive stalls. Cover all the necessary stalls, while covering as few total stalls as possible.

## The Idea

The basic idea behind greedy algorithms is to build large solutions up from smaller ones. Unlike other approaches, however, greedy algorithms keep only the best solution they find as they go along. Thus, for the sample problem, to build the answer for N = 5, they find the best solution for N = 4, and then alter it to get a solution for N = 5. No other solution for N = 4 is ever considered.

Greedy algorithms are **fast**, generally linear to quadratic and require little extra memory. Unfortunately, they usually aren't correct. But when they do work, they are often easy to implement and fast enough to execute.

## Problems

There are two basic problems to greedy algorithms.

### How to Build

How does one create larger solutions from smaller ones? In general, this is a function of the problem. For the sample problem, the most obvious way to go from four boards to five boards is to pick a board and remove a section, thus creating two boards from one. You should choose to remove the largest section from any board which covers only stalls which don't need covering (so as to minimize the total number of stalls covered).

To remove a section of covered stalls, take the board which spans those stalls, and make into two boards: one of which covers the stalls before the section, one of which covers the stalls after the section.

### Does it work?

The real challenge for the programmer lies in the fact that greedy solutions don't always work. Even if they seem to work for the sample input, random input, and all the cases you can think of, if there's a case where it won't work, at least one (if not more!) of the judges' test cases will be of that form.

For the sample problem, to see that the greedy algorithm described above works, consider the following:

Assume that the answer doesn't contain the large gap which the algorithm removed, but does contain a gap which is smaller. By combining the two boards at the end of the smaller gap and splitting the board across the larger gap, an answer is obtained which uses as many boards as the original solution but which covers fewer stalls. This new answer is better, so therefore the assumption is wrong and we should always choose to remove the largest gap.

If the answer doesn't contain this particular gap but does contain another gap which is just as large, doing the same transformation yields an answer which uses as many boards and covers as many stalls as the other answer. This new answer is just as good as the original solution but no better, so we may choose either.

Thus, there exists an optimal answer which contains the large gap, so at each step, there is always an optimal answer which is a superset of the current state. Thus, the final answer is optimal.

## Conclusions

If a greedy solution exists, use it. They are easy to code, easy to debug, run quickly, and use little memory, basically defining a good algorithm in contest terms. The only missing element from that list is correctness. If the greedy algorithm finds the correct answer, go for it, but don't get suckered into thinking the greedy solution will work for all problems.

## Sample Problems

#### Sorting a three-valued sequence [IOI 1996]

You are given a three-valued (1, 2, or 3) sequence of length up to 1000. Find a minimum set of exchanges to put the sequence in sorted order.

**Algorithm** The sequence has three parts: the part which will be 1 when in sorted order, 2 when in sorted order, and 3 when in sorted order. The greedy algorithm swaps as many as possible of the 1's in the 2 part with 2's in the 1 part, as many as possible 1's in the 3 part with 3's in the 1 part, and 2's in the 3 part with 3's in the 2 part. Once none of these types remains, the remaining elements out of place need to be rotated one way or the other in sets of 3. You can optimally sort these by swapping all the 1's into place and then all the 2's into place.

Analysis: Obviously, a swap can put at most two elements in place, so all the swaps of the first type are optimal. Also, it is clear that they use different types of elements, so there is no ``interference'' between those types. This means the order does not matter. Once those swaps have been performed, the best you can do is two swaps for every three elements not in the correct location, which is what the second part will achieve (for example, all the 1's are put in place but no others; then all that remains are 2's in the 3's place and vice-versa, and which can be swapped).

#### Friendly Coins - A Counterexample [abridged]

Given the denominations of coins for a newly founded country, the Dairy Republic, and some monetary amount, find the smallest set of coins that sums to that amount. The Dairy Republic is guaranteed to have a 1 cent coin.

Algorithm: Take the largest coin value that isn't more than the goal and iterate on the total minus this value.

(Faulty) Analysis: Obviously, you'd never want to take a smaller coin value, as that would mean you'd have to take more coins to make up the difference, so this algorithm works.

Maybe not: Okay, the algorithm usually works. In fact, for the U.S. coin system {1, 5, 10, 25}, it always yields the optimal set. However, for other sets, like {1, 5, 8, 10} and a goal of 13, this greedy algorithm would take one 10, and then three 1's, for a total of four coins, when the two coin solution {5, 8} also exists.

**Topological Sort**

Given a collection of objects, along with some ordering constraints, such as "A must be before B," find an order of the objects such that all the ordering constraints hold.

Algorithm: Create a directed graph over the objects, where there is an arc from A to B if "A must be before B." Make a pass through the objects in arbitrary order. Each time you find an object with in-degree of 0, greedily place it on the end of the current ordering, delete all of its out-arcs, and recurse on its (former) children, performing the same check. If this algorithm gets through all the objects without putting every object in the ordering, there is no ordering which satisfies the constraints.

USACO Gateway | Comment or Question

# Heuristic Search

## Prerequisite

- Recursive Descent

## Main Concept

The main goal of *heuristic search* is to use an *estimate* of the ``goodness'' of all states to improve the search for a solution.

Generally, this estimate of ``goodness'' is expressed as a function of the state, and such functions are called *heuristic functions*. Examples of potential heuristic functions are as follows:

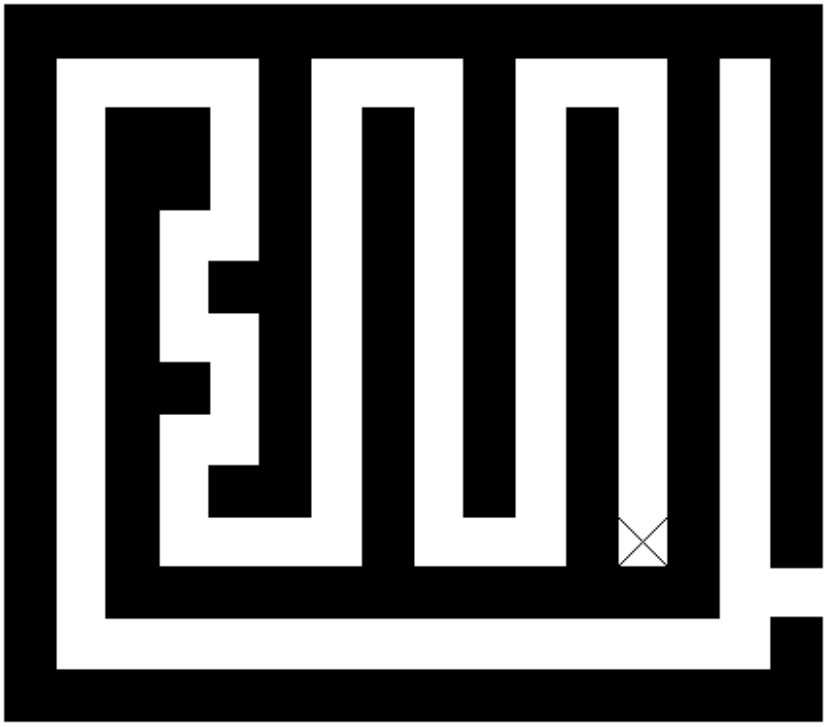
- When looking for an exit in a maze, the euclidean distance to the exit.
- When trying to win a game of checkers, the number of checkers you have minus the number of checkers your opponent has.
- When trying to win freecell, the number of cards already home plus 5 times the number of freecells that are empty.

## Designing Heuristic Functions

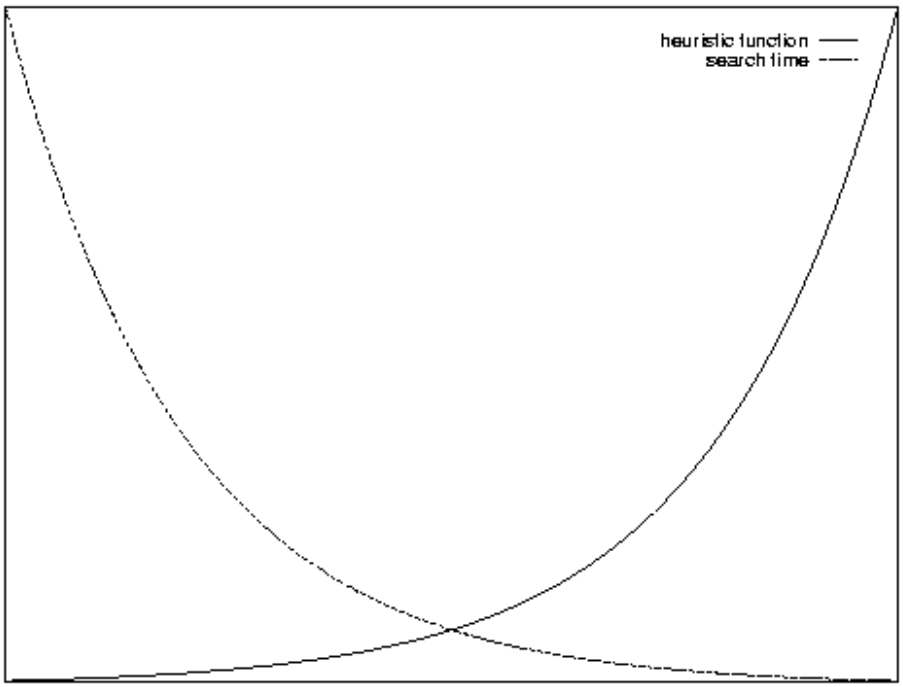
Intuitively, the better a heuristic function is, the better (faster) a search will be. The question is: how does one evaluate how good a heuristic function is?

A heuristic function is evaluated in terms of how well it estimates the cost of a state. For example, for the case of a maze, how well does it estimate the distance to the exit? Euclidean distance can be very bad, as even for simple mazes, it can be far off from the

same solution:



In general, the better the heuristic function, the faster the search will go. In general, the search time and accuracy of the heuristic function behave in the following manner.




It's important to notice that even a stupid heuristic function can significantly improve the search (if it's used properly of course).

Another concept is important when looking for a heuristic function, *admissibility*. A heuristic function is called *admissible* if it *underestimates* the cost of a state and is nonnegative for all states. For example, the euclidean distance for the maze is an underestimate, as the example above clearly shows.

### Idea #1: Heuristic Pruning

The easiest and most common use for heuristic functions is to prune the search space. Assume the problem is to find the solution with the minimum total cost. With an admissible heuristic function, if the cost of the current solution thus far is A, and the heuristic function returns B, then the best possible solution which is a child of the current solution is A+B. If the a solution has been found with cost C, where C < A+B, there is no reason to continue searching for a solution from this state.

This is simple to code and debug (assuming one starts with a working, but slow, program, which is how this should be used) and can yield *immense* returns in terms of run-time. It is especially helpful with depth-first search with iterative deepening.

### Idea #2: Best-First Search

The way to think of best-search is as greedy depth-first search.

Instead of expanding the children in an arbitrary order, a best-first search expands them in order of their ``goodness,'' as defined by the heuristic function. Unlike greedy search, which *only* tries the most promising path, best-first search tries the most promising path first, but later tries less promising ones. When combined with the pruning described above, this can yield very good results.

### Idea #3: A* Search

The *A* Search* is akin to the greedy breadth-first search.

Breadth-first search always expands the node with minimum cost. A* search, on the other hand, expands the node which looks the most promising (that is, the cost of reaching that state plus the heuristic value of that state is minimum).

The states are kept in a priority queue, with their priority the sum of their cost plus the heuristic evaluation. At each step, the algorithm removes the lowest priority item and places all of its children into the queue with the appropriate priority.

With an admissible heuristic function, the first end state that A* finds is guaranteed to be optimal.

### Example Problems

#### Knight Cover [Traditional]

Place as few knights as possible on an *n x n* chess board so that every square is attacked. A knight is not considered to attack the square on which it sits.

Analysis: One possible heuristic function is the total number of squares attacked divided by eight (the number of squares a knight can attack). This can be used for any of the forms, although A* has problems, as the queue becomes large.

### 8-puzzle [Traditional]

Given a 3 x 3 grid of numbers, with one square free, you are allowed to move any number adjacent to the blank square from its current location to the blank square.

```
5 _ 4    5 1 4
6 1 8    6 _ 8
7 3 2    7 3 2
```

What is the minimum number of moves required to get back to the following state (you are guaranteed to be able to):

```
1 2 3
4 5 6
7 8 _
```

Heuristic function: The `taxi-cab' distance between each number and the location it's supposed to be in.

Analysis: Because the state space is fairly small (only 362,880), A* will work well, if you ensure no state is ever in the queue twice, and no visited state is ever put into the queue.

USACO Gateway | Comment or Question

# Knapsack Problems

## Prerequisite modules

- Greedy
- Dynamic Programming
- Recursive Descent

## Sample Problem: Tape Recording

Farmer John's favorite hobby is creating a tape containing some of Bessie's favorite music to listen to while she's being milked. The amount of milk that Bessie produces is dependent on the songs that Bessie listens to while being milked.

Given a collection of songs, each represented by a pair of integers, the length of the song (in seconds), the amount of milk Bessie produces while she's listening to that song, and the total amount of time that it takes to milk Bessie, find the set of songs such that their total length is no more than the time it takes to milk Bessie and they maximize Bessie's milk production.

## The Abstraction

Given, A collection of objects, each which a size, a value (i.e., weight), and the total `space' available, find the set of objects which maximizes the sum of the value of the set, but whose sum of size is constrained by some limit. The total number/size of any particular item used in the set cannot exceed its availability.

## Problem Viewpoint

The general way to view a knapsack problem is that of a bag of limited capacity, which is to be filled while maximizing the value of the objects in it.

For the problem above, the tape which Bessie will listen to while being milked is the ``knapsack,'' while the songs are the ``objects to be placed within the knapsack.''

## Three Knapsack Problems

The knapsack problem style has three forms:

- Fractional knapsack problem
  A fractional knapsack problem is one in which you are allowed to place fractional objects in the knapsack. For example, if the objects were crude oil, airplane fuel, and kerosene and your knapsack a bucket, it might make sense to take 0.473 liter of the crude oil, 0.263 liter of the airplane fuel, and 0.264 liter of the kerosene. This is the easiest form of the knapsack problem to solve.

- Integer Knapsack problem
  In integer knapsack problems, only complete objects can be inserted into the knapsack. The example problem is of this form: partial songs aren't allowed.
- Multiple knapsack problem
  In the multiple knapsack problem, more than one knapsack is to be filled. If fractional objects are allowed, this is the same as having one large knapsack with capacity equal to the sum of all the available knapsacks, so this term will only be used to refer to the case of multiple integer knapsacks.

**Fractional knapsack problem**

The fractional knapsack problem is the easiest of the three to solve, as the greedy solution works:

- Find the object which has the highest ``value density'' (value of object / size).
- If the total amount of capacity remaining exceeds the availability of that object, put all of it in the knapsack and iterate.
- If the amount of capacity is less than the availability of the object, use as much as possible and terminate.
- This algorithm runs in *N log N* since it must sort the objects first based on value density and then put them into the knapsack in decreasing order until the capacity is used. It's normally easier to not sort them but rather just keep finding the highest value density not used each time, which gives a $O(N^2)$ algorithm.

Side note: For problems of this class, it's rare to have both size and availability, as you can do a trivial transformation to have all the objects of size 1, and the availability be the product of the original size and availability (dividing the value by the original size, of course).

Extensions: The value and availability of the objects can be real numbers without a problem in this case. The fractional size issue is also trivial to handle by this algorithm.

**Integer knapsack problem**

This is slightly more difficult, but is solvable using dynamic programming if the knapsack is small enough.

- Do dynamic programming on the maximum value that a knapsack of each size can have in it.
- Update this array for an object of size S by traversing the array in reverse order (of capacity), and seeing if placing the current object into the knapsack of size K yields a better set than the current best knapsack of size K+S.
- This algorithm runs in *K x N* time, where *K* is the size of the knapsack, and *N* is the sum of availability of objects.
- If the knapsack is too large to allocate this array, recursive descent is the method of choice, as this problem is NP-complete. Of course, recursive descent can run for a very long time in a large knapsack being filled with small objects.

Extensions:

- Fractional values are not a problem; the array just becomes an array of real numbers instead of integers. Fractional availability doesn't affect things, as you can,

without loss of generality, truncate the number (if you have 3.5 objects, you can only use 3).
- Fractional size is a pain, as it makes the problem recursive descent.
- If the sizes are all the same, the problem can be solved greedily, picking the objects in decreasing value order until the knapsack is full.
- If the values are all 1.0, then again greedy works, selecting the objects in increasing size order until the knapsack is full.

## Multiple knapsack problem

With multiple knapsacks of any size, the state space is too large to use the DP solution from the integer knapsack algorithm. Thus, recursive descent is the method to solve this problem. Extensions:

- With recursive descent, extensions are generally easy. Fractional sizes and values are no problem, nor is another evaluation function.
- If the values are all one, then if the maximum number of objects that can be placed in all the knapsacks is n, then there is such a solution which uses the n smallest objects. This can greatly reduce the search time.

## Sample Problems

#### Score Inflation [1998 USACO National Championship]

You are trying to design a contest which has the maximum number of points (<10,000). Given the length of the contest, a group of problems, the problem lengths, and the point value of each problem, find the contest which has the maximum number of points (which satisfies the length constraint).

Analysis: This is an integer knapsack problem, where the knapsack is the contest, the sizes are the length of problems, and the values are the point values. The limit on knapsack (contest) size is small enough that the DP solution will work in memory.

#### Fence Rails [1999 USACO Spring Open]

Farmer John is trying to construct a fence around his field. He has installed the posts already, so he knows the length requirement of the rails. The local lumber store has dropped off some boards (up to 50) of varying length. Given the length of the boards from the lumber store, and the lengths of rails that Farmer John needs, calculate the maximum numbers of rails that Farmer John can create.

Analysis: This is a multiple knapsack problem, where the knapsacks are the boards from the store, and the objects are the rails that Farmer John needs. The size of the objects are just the length, and the value is just one.

Since the values are all one, you know that if there is a solution using any K rails, there is a solution using the K smallest rails, which helps the recursive descent solver quite a bit.

#### Filling your Tank

You're in the middle of Beaver County, at the only city within 100 miles with a gas station, trying to fill up your tank so you can get to Rita Blanca. Fortunately, this town has a couple of gas stations, but they all seem to be almost out of gas. Given the price of gasoline at each station, and the amount of gas each one has, calculate how much gasoline to buy from each station in order to minimize the total cost.

Analysis: This is an fractional knapsack problem, where your knapsack is your gas tank, and the objects are gasoline.

USACO Gateway | Comment or Question

# Minimum Spanning Tree

### Sample Problem: Agri-Net [Russ Cox, Winter 1999 USACO Open]

Farmer John is bringing internet connectivity to all farms in the area. He has ordered a high speed connection for his farm and is going to share his connectivity with the other farmers. To minimize cost, he wants to minimize the length of optical fiber to connect his farm to all the other farms.

Given a list of how much fiber it takes to connect each pair of farms, find the minimum amount of fiber needed to connect them all together. Each farm must connect to some other farm such that a path exists from any farm to any other farm. Some farms might have 1, 2, 3, or more connections to them.

### The Abstraction

Given: an undirected, connected graph with weighted edges

A *spanning tree* of a graph is any sub-graph which is a connected tree (i.e., there exists a path between any nodes of the original graph which lies entirely in the sub-graph).

A *minimal* spanning tree is a spanning tree which has minimal `cost' (where cost is the sum of the weights of the edges in the tree).

### Prim's algorithm to construct a Minimal Spanning Tree

Given: lists of nodes, edges, and edge costs

The algorithm (greedily) builds the minimal spanning tree by iteratively adding nodes into a working tree.

- Start with a tree which contains only one node. Iteratively find the closest node to that one and add the edge between them.
- Let the distance from each node not in the tree to the tree be the edge (connection) of minimal weight between that node and some node in the tree. If there is no such edge, then assume the distance is infinity (this shouldn't happen).
- At each step, identify a node (outside the tree) which is closest to the tree and add the minimum weight edge from that node to some node in the tree and incorporate the additional node as a part of the tree.

For analysis of why this works, consult Chapter 24 of [Cormen, Leiserson, Rivest].

Here is pseudocode for the algorithm:

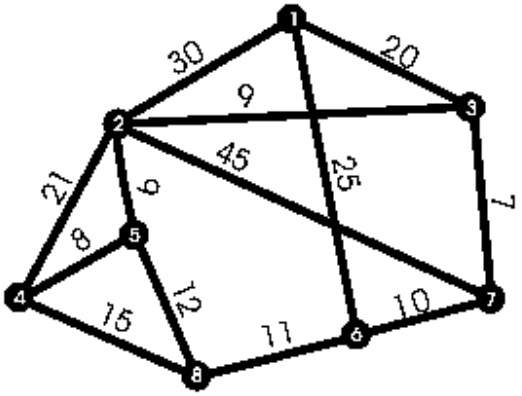
```
  # distance(j) is distance from tree to node j
  # source(j) is which node of so-far connected MST
  #                    is closest to node j
1   For all nodes i
2     distance(i) = infinity        # no connections
3     intree(i) = False             # no nodes in tree
```

```
 4     source(i) = nil

 5   treesize = 1                    # add node 1 to tree
 6   treecost = 0
 7   intree(1) = True
 8   For all neighbors j of node 1   # update distances
 9      distance(j) = weight(1,j)
10     source(j) = 1

11   while (treesize < graphsize)
12     find node with minimum distance to tree; call it node i
13     assert (distance(i) != infinity, "Graph Is Not Connected")

     # add edge source(i),i to MST
14     treesize = treesize + 1
15     treecost = treecost + distance(i)
16     intree(i) = True              # mark node i as in tree

     # update distance after node i added
17     for all neighbors j of node i
18       if (distance(j) > weight(i,j))
19         distance(j) = weight(i,j)
20         source(j) = i
```

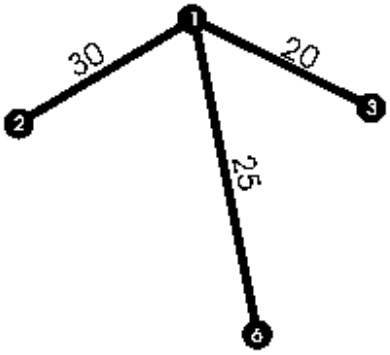
Running time of this formulation is O($N^2$). You can obtain O($N$ log $N$) for sparse graphs, but it normally isn't worth the extra programming time.

### Execution Example

Consider the following graph with weighted edges:




The goal is to find the minimal spanning tree. The algorithm will start at node 1 which connects to nodes 2, 6, and 3 with the weights shown on the edges:

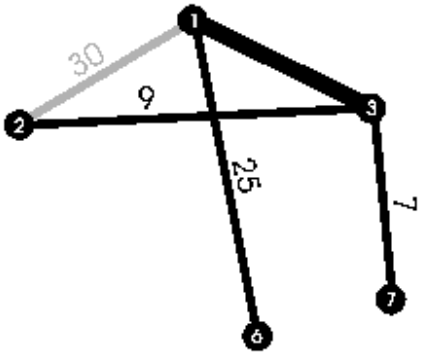



| Node | distance | intree | source |
|---|---|---|---|
| 1 | infinity | True | nil |
| 2 | 30 | False | 1 |
| 3 | 20 | False | 1 |

| | | | |
|---|---|---|---|
| 6 | 25 | False | 1 |

All nodes not shown have infinite distance, intree=False, and source=nil.

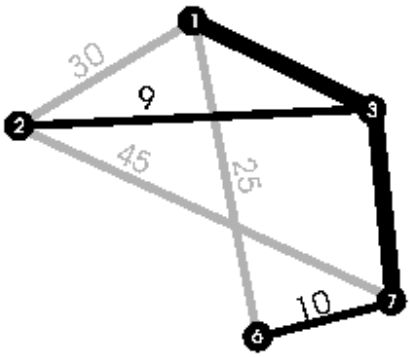
The smallest distance **for a node not in the tree** is 20, so the listed edge to node 3 is added to the tree:




| Node | distance | intree | source |
|---|---|---|---|
| 1 | infinity | True | nil |
| 2 | **9** | False | **3** |
| 3 | 20 | **True** | 1 |
| 6 | 25 | False | 1 |
| **7** | 7 | False | 3 |

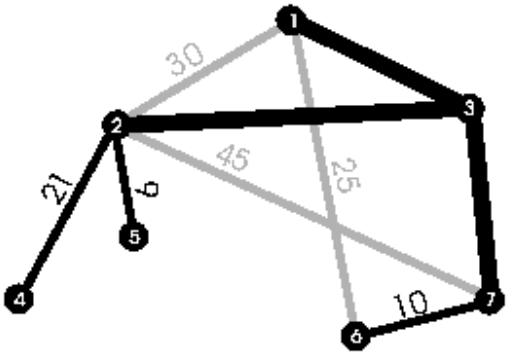
Note that node 3 is now `in the tree'. Node 2's distance changed to 9 while the source changed to 3.

The smallest distance is 7, so the edge from node 3 to node 7 (coincidental name!) is connected:



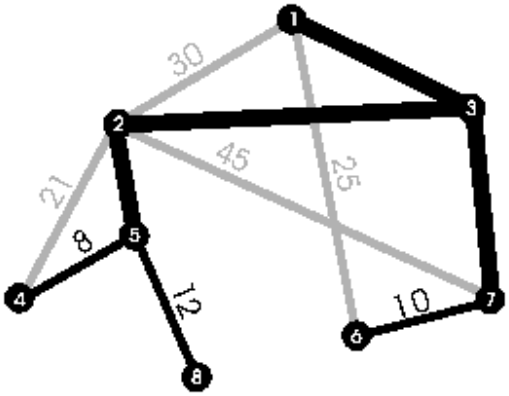

| Node | distance | intree | source |
|---|---|---|---|
| 1 | infinity | True | nil |
| 2 | 9 | False | 3 |
| 3 | 20 | True | 1 |
| 6 | **10** | False | **7** |
| 7 | 7 | **True** | 3 |

Node 2's distance is 9, the smallest of any node not in the tree. Adding the edge from node 3 to node 2 results in a graph that looks like this:



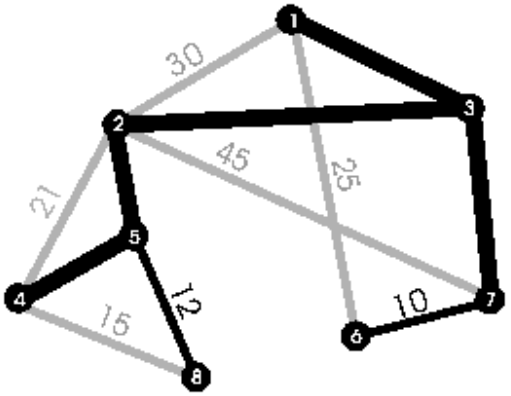

| Node | distance | intree | source |
|---|---|---|---|
| 1 | infinity | True | nil |
| 2 | 9 | **True** | 3 |
| 3 | 20 | True | 1 |
| **4** | 21 | False | 2 |
| **5** | 9 | False | 2 |
| 6 | 10 | False | 7 |
| 7 | 7 | True | 3 |

Add the edge from node 2 to node 5:



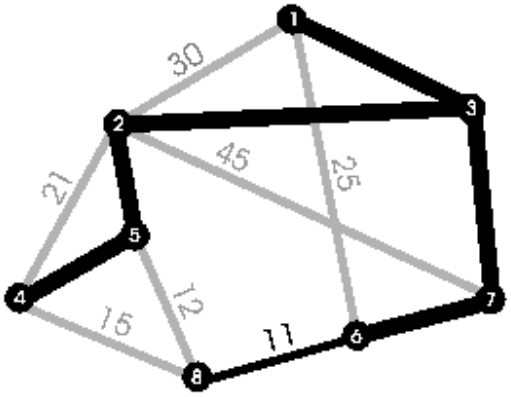

| Node | distance | intree | source |
|---|---|---|---|
| 1 | infinity | True | nil |
| 2 | 9 | **True** | 3 |
| 3 | 20 | True | 1 |
| 4 | **8** | False | **5** |
| 5 | 9 | **True** | 2 |
| 6 | 10 | False | 7 |
| 7 | 7 | True | 3 |
| **8** | 12 | False | 5 |

Adding the edge from node 5 to node 4 is the next step:



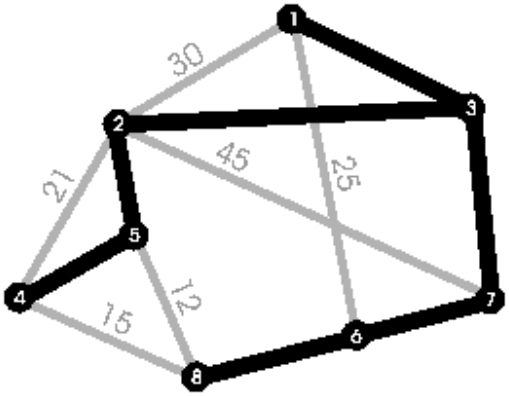

| Node | distance | intree | source |
|---|---|---|---|
| 1 | infinity | True | nil |
| 2 | 9 | True | 3 |
| 3 | 20 | True | 1 |
| 4 | 8 | **True** | 5 |
| 5 | 9 | True | 2 |
| 6 | 10 | False | 7 |
| 7 | 7 | True | 3 |
| **8** | 12 | False | 5 |

Next up: edge connecting nodes 6 and 7:




| Node | distance | intree | source |
|---|---|---|---|
| 1 | infinity | True | nil |
| 2 | 9 | True | 3 |
| 3 | 20 | True | 1 |
| 4 | 8 | True | 5 |
| 5 | 9 | True | 2 |
| 6 | 10 | **True** | 7 |
| 7 | 7 | True | 3 |
| 8 | **11** | False | **6** |

Finally, add the edge from node 6 to node 8:




| Node | distance | intree | source |
|---|---|---|---|
| 1 | infinity | True | nil |
| 2 | 9 | True | 3 |
| 3 | 20 | True | 1 |
| 4 | 8 | True | 5 |
| 5 | 9 | True | 2 |
| 6 | 10 | True | 7 |
| 7 | 7 | True | 3 |
| 8 | 11 | **True** | 6 |

And the minimal spanning tree is easily seen here.

**Dangerous Curve**

Understand that changing any element in a tree requires complete recalculation - incremental recalculation of a spanning tree when changing isolated nodes, for example, is not generally possible.

**Problem Cues**

If the problem mentions wanting an optimal, connected sub-graph, a minimum cost way to connect a system together, or a path between any two parts of the system, it is very likely to be a minimum spanning tree problem.

**Extensions**

If you subject the tree to any other constraints (no two nodes may be very far away or the average distance must be low), this algorithm breaks down and altering the program to handle such constraints is very difficult.

There is obviously no problem with multiple edges between two nodes (you ignore all but the smallest weight).

Prim's algorithm does not extend to directed graphs (where you want strong connectedness), either.

## Sample Problems

**Package Routing**

Given: a set of locations of cities and the cost of connecting each pair of cities for a shipping company. Find the cheapest set of pairs of cities such that a package can be routed from any city to any other city.

**Highway Building**

Lower Slobbovia has made the plunge and has decided to connect all their cities with roads. Of course, being cheap, they want to spend as little money as possible. The cost of a highway is linearly proportional to its length. Given the x,y coordinates of the cities in L.S., find the cheapest way to interconnect the cities.

**Bovile Phones (abridged) [USACO Training Camp 1998, Contest 2]**

Given: a collection of stationary cows and haystacks in the field along with a cost function for connecting two (arbitrary) locations. Using only the haystacks and cows, calculate which haystacks one should include in a network to minimize the total cost.

Analysis: For each possible set of haystacks (i.e., about $2^n$ sets), calculate the cost of the minimal spanning tree of the haystacks in that set along with all the cows. Find the combination of haystacks that minimizes the cost.

USACO Gateway | Comment or Question

# Network Flow Algorithms

## Prerequisite

- Shortest Path

## The Problem

Given: A direct connected graph with integer weighted arcs, along with a source node and a sink node.

Each arc weight corresponds to the ``capacity'' of that arc. A flow through the graph is constructed by assigning an integer amount of ``flow'' to send through each edge such that:

- The flow through each arc is no greater than the arc's capacity.
- For each node other than the source and sink, total flow in is the same as total flow out.

Maximize the total of the weights of the out-arcs of the source minus the weights of the in-arcs (or the total of the weights of the in-arcs of the sink minus the weights of the out-arcs).

### Example

Given: The layout of a collection of water pipes, along with the capacity of each pipe. Water in these pipes must flow downhill, so within each pipe, water can only flow in one direction.

Calculate the amount of water that can flow from a given start (the water-purification plant) to a given end (your farm).

## The Algorithm

The algorithm (greedily) builds the network flow by iteratively adding flow from the source to the sink.

Start with every arc having weight equal to the beginning weight (The arc weights will correspond to the amount of capacity still unused in that arc).

Given the current graph, find a path from the source to the sink across arcs that all have non-zero weight in the current graph. Calculate the maximum flow across this path, call it PathCap.

For each arc along the path, reduce the capacity of that arc by PathCap. In addition, add the reverse arc (the arc between the same two nodes, but in the opposite direction) with capacity equal to PathCap (if the reverse arc already exists, just increase its capacity).

Continue to add paths until none exist.

This is guaranteed to terminate because you add at least one unit of flow each time (since the weights are always integers), and the flow is strictly monotonically increasing. The use of an added reverse arc is equivalent to reducing the flow along that path.

If you are interested in a more detailed analysis of this algorithm, consult Sedgewick.

Here is pseudocode for the algorithm:

```
 1   if (source = sink)
 2     totalflow = Infinity
 3     DONE

 4   totalflow = 0

 5   while (True)
 6 # find path with highest capacity from
   # source to sink
 7 # uses a modified djikstra's algorithm
 8     for all nodes i
 9       prevnode(i) = nil
10       flow(i) = 0
11       visited(i) = False
12     flow(source) = infinity

13     while (True)
14       maxflow = 0
15       maxloc = nil
16       # find the unvisited node with
         # the highest capacity to it
17       for all nodes i
18         if (flow(i) > maxflow AND
                         not visited(i))
19           maxflow = flow(i)
20           maxloc = i
21       if (maxloc = nil)
22         break inner while loop
23       if (maxloc = sink)
24         break inner while loop
24a      visited(maxloc) = true
25       # update its neighbors
26       for all neighbors i of maxloc
27         if (flow(i) < min(maxflow,
                     capacity(maxloc,i)))
28           prevnode(i) = maxloc
29           flow(i) = min(maxflow,
                    capacity(maxloc,i))

30     if (maxloc = nil)         # no path
31       break outer while loop

32     pathcapacity = flow(sink)
33     totalflow = totalflow + pathcapacity

   # add that flow to the network,
   # update capacity appropriately
35     curnode = sink
         # for each arc, prevnode(curnode),
         # curnode on path:
36     while (curnode != source)
38       nextnode = prevnode(curnode)
39       capacity(nextnode,curnode) =
           capacity(nextnode,curnode) -
40                          pathcapacity
41       capacity(curnode,nextnode) =
           capacity(curnode,nextnode) +
42                          pathcapacity
43       curnode = nextnode
```

Running time of this formulation is O(F M), where F is the maximum flow and M is the number of arcs. You will generally perform much better, as the algorithm adds as much flow as possible every time.

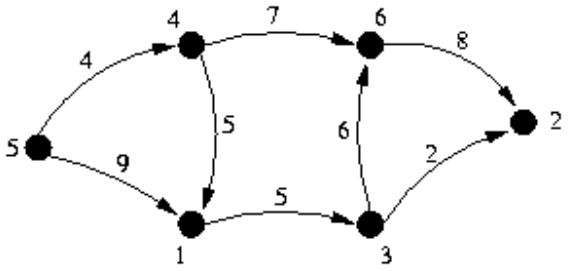
To determine the flow across each arc, compare the starting capacity with the final capacity. If the final capacity is less, the difference is the amount of flow traversing that

arc.

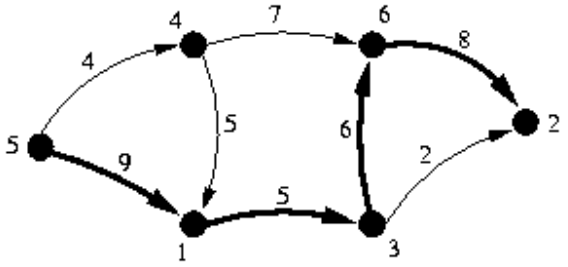
This algorithm may create `eddies,' where there is a loop which does not contribute to the flow itself.

**Execution Example**

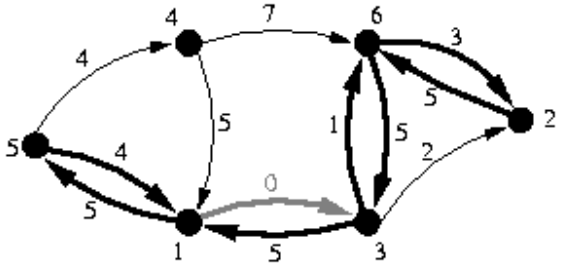
Consider the following network, where the source is node 5, and the sink is node 2.



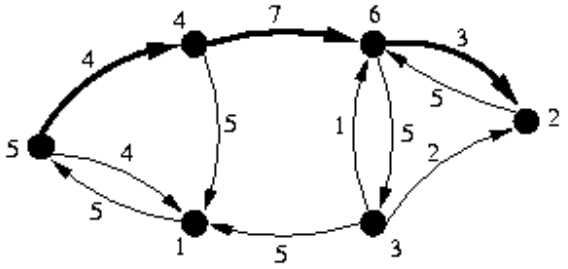

The path with the highest capacity is {5,1,3,6,2}.



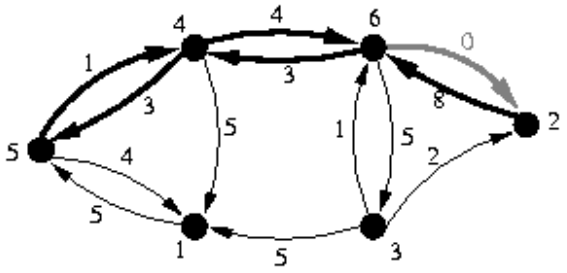

The bottleneck arc on this path is 1->3, which has a capacity of 5. Thus, reduce all arcs on the path by 5, and add 5 to the capacity of the reverse arcs (creating the arcs, if necessary). This gives the following graph:



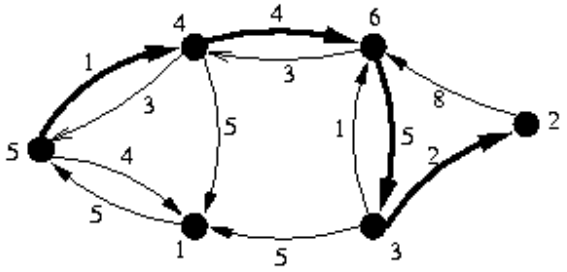

In the new graph, the path with highest capacity is {5,4,6,2}.



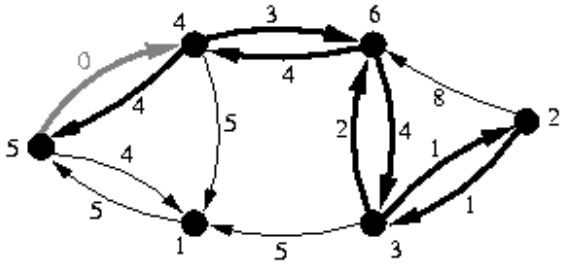

The capacity of this path is 3, so once again, reduce the forward arcs by 3, and increase the reverse arcs by 3.



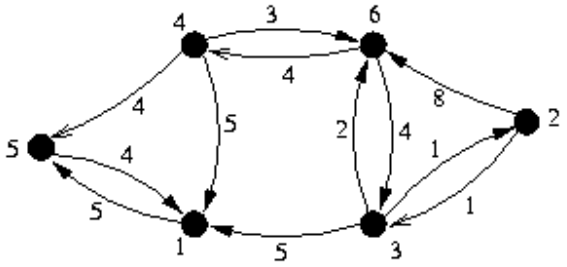

Now the network's maximum capacity path is {5,4,6,3,2}




This flow has only a capacity of 1, as the arc from 5 to 4 has capacity 1. Once again, update the forward and backwards arcs appropriately.




The resulting graph has no paths from the source to the sink. The only nodes reachable from the source node 5 are node 5 itself and node 1.




The algorithm added three flows, the first with capacity 5, the second with capacity 3, and the last with capacity 1. Thus, the maximum flow through the network from node 5 to node 2 is 9.

**Extensions**

Network flow problems are very extensible, mostly by playing with the graph.

To extend to the case of undirected graphs, simple expand the edge as two arcs in opposite directions.

If you want to limit the amount of traffic through any node, split each node into two nodes, an in-node and and out-node. Put all the in-arcs into the in-node, and all of the

out-arcs out of the out-node and place an arc from the in-node to the out-node with capacity equal to the capacity of the node.

If you have multiple sources and sinks, create a `virtual source' and `virtual sink' with arcs from the virtual source to each of the sources and arcs from each of the sinks to the virtual sink. Make each of the added arcs have infinite capacity.

If you have arcs with real-valued weights, then this algorithm is no longer guaranteed to terminate, although it will asymptotically approach the maximum.

## Alternative Problems

Network flow can also be used to solve other types of problems that aren't so obvious

#### Maximum Matching

Given a two sets of objects (call them A and B), where you want to `match' as many individual A objects with individual B objects as possible, subject to the constraint that only certain pairs are possible (object A1 can be matched with object B3, but not object B1 or B2). This is called the `maximum matching' problem.

To reformulate this as network flow, create a source and add an arc with capacity 1 from this source to each A object. Create a sink with an arc from each B object to it with capacity 1. In addition, if object Ai and Bk may be matched together, add an arc from Ai to Bk with capacity 1. Now run the algorithm and determine which arcs between A objects and B objects are used.

#### Minimum Cut

Given a weight undirected graph, what is the set of edges with minimum total weight such that it separates two given nodes.

The minimum total weight is exactly the flow between those two nodes.

To determine the path, try removing each edge in increasing weight order, and seeing if it reduces the network flow (if it does, it should reduce the flow by the capacity of that edge. The first one which does is a member of the minimum cut, iterate on the graph without that edge.

This can be extended to node cuts by the same trick as nodes with limited capacity. Directed graphs work using the same trick. However, it can not solve the problem of finding a so-called `best match,' where each pairing has a `goodness' value, and you want to create the matching which has the highest total `goodness.'

## Example Problems

If the problems talks about maximizing the movement or flow of something from one location to another, it is almost assuredly maximum flow. If it talks about trying to separate two items minimally, it is probably minimum cut. If it talks about maximizes the pairing of any sort of thing (person, object, money, whatever), it is probably maximum matching.

#### Virus Flow

You have a computer network, with individual machines connected together by wires. Data may flow either direction on the wire. Unfortunately, a machine on your network has caught a virus, so you need to separate this machine from your central server to stop the spread of this virus. Given the cost of shutting down the network connection between each pair of machines, calculate the minimum amount of money which must be spent to separate the contaminated machine from your server.

This is exactly the min cut problem.

**Lumberjack Scheduling**

Different types of trees require different techniques to be employed by lumberjacks for them to harvest the tree properly. Regardless of the tree or lumberjack, harvest a tree requires 30 minutes. Given a collection of lumberjacks, and the types of trees that each one is able to correctly cut down, and a collection of trees, calculate the maximum number of trees which may be cut down in the next half hour.

Each lumberjack can be paired with each tree of a type that he/she is able to properly harvest. Thus, the problem can be solved using the maximum matching algorithm.

**Telecowmunication (USACO Championship 1996)**

Given a group of computers in the field, along with the wires running between the computers, what is the minimum number of machines which may crash before two given machines are the network are unable to communicate? Assume that the two given machines will not crash.

This is equivalent to the minimum node cut problem. The two given machines can be arbitrarily labeled the source and sink. The wires are bidirectional. Split each node into an in-node and an out-node, so that we limit the flow through any given machine to 1. Now, the maximum flow across this network is equivalent to the minimum node cut.

To actually determine the cut, iterative remove the nodes until you find one which lowers the capacity of the network.

**Science Fair Judging**

A science fair has N categories, and M judges. Each judge is willing to judge some subset of the categories, and each category needs some number of judges. Each judge is only able to judge one category at a given science fair. How many judges can you assign subject to these constraints?

This is very similar to the maximum matching problem, except that each category can handle possibly more than one judge. The easiest way to do this is to increase the capacity of the arcs from categories to the sink to be the number of judges required.

**Oil Pipe Planning**

Given the layout (the capacity of each pipe, and how the pipes are connected together) of the pipelines in Alaska, and the location of each of the intersections, you wish to increase the maximum flow between Juneau and Fairbanks, but you have enough money to only add one pipe of capacity X. Moreover, the pipe can only be 10 miles long.

Between which two intersections should this pipe be added to increase the flow the most?

To solve this problem, for each pair of intersections within 10 miles of each other, calculate the increase in the flow between Juneau and Fairbanks if you add a pipe between the intersections. Each of these sub-problems is exactly maximum flow.

USACO Gateway | Comment or Question

# Optimization

## What is Optimization?

### Making Working Code Faster

Note *working*. Until a program works, do not try to make it faster, as debugging is then more complicated. Also, before making any change for optimization reasons (instead of debugging reasons), make a backup of the working code. There's nothing more painful then figuring out three optimizations deep that the first optimization broke the program, and no backup exists.

This module will focus on ways to speed programs without altering the program's algorithm. If there's a faster algorithm, that's where effort should be focused, not on making a slow algorithm run a little less slow, which is much like putting perfume on a pig.

In particular, this module will only discuss ways to make a recursive descent program run more quickly, by trying to avoid searching the entire tree. This methodology is referred to as *pruning* the search.

## A Problem: Chains

This is a slight adaptation of a real-world problem in compilers, although in that world, only multiplication is available.

Suppose a program must find $x^n$ exactly (no log/exp) and the two standard operations, multiplication and divide, are available. What is the minimal number of these operations required? Show the list (in order) of powers of $x$ that can be calculated in order to find $x^n$.

For this explication, the calculations will be represented in terms only of the exponent. Multiplication will be written as addition (or the sum of two previous powers), division as subtraction (or the difference of two previous powers). For legal chains, each entry must be either the sum or difference of some two previous entries. So, calculating $x^8$ would be represented as: 1 2 4 8 (three entries beyond the initial `1'); calculating $x^{15}$ might be represented as: 1 2 4 8 16 15.

### Test Set #1

Consider the data set requiring successive calculation of all the possible powers for the exponents 1..50 (not all at once - 50 different test cases). All timings are for a 233 MHz Pentium II.

### A Basic Algorithm

Start with the set {1}, perform a depth-first search on generatable numbers (Note that to make this even close to reasonable, the pairs will be chosen in reverse order, that is start with adding the last number in the current sequence to itself, then adding to the previous number, then the difference between the last two in the sequence, etc.)

#### What's Wrong With This?

It will never terminate.

So, change the algorithm to prune when considering a list that is longer than the best list found thus far. Assume the maximum length chain is 32 steps (a true statement, for exponents up to 65536, but unproven; the assumption will be removed later).

Runtime for sample implementation: 4658.34 seconds.

## Optimization Basics

#### Prune Early, Prune Often

Consider: in a tree of fan-out only 2, getting rid of just two levels everywhere on that tree reduces the number of nodes by a factor of almost 4.

Two basic ideas signal the way to make searching programs faster:

- Don't Do Anything Stupid
- Don't Do Anything Twice

The problem is figuring out what is being done twice and what steps are stupid. All of the optimizations discussed here will utilize some basic fact of the search space. Make sure that the fact is true!

## Improving Chain Production

#### Observation #1

There is one easy way to come up with the numbers, or at least an upper bound. Examine the binary representation of the number. Produce all the powers of 2 up to *n*, and then add up those that that the binary representation suggests. (It is possible to do a little better than this using subtractions, but that won't matter in the long run). Use this scheme to initialize the ``best answer'' data, so time is not spent searching for very long answers.

For example, 43 is 101011 in base 2. Thus, the sequence would start 1, 2, 4, 8, 16, 32. Now, add the ones in the base 2 representation, so 1 + 2 = 3 would be the next number, then 3 + 8 = 11, then 11 + 32 = 43. This yields a sequence of length 9: 1, 2, 4, 8, 16, 32, 3, 11, 43.

Runtime: 4609.81 seconds.

Evaluation: This is just barely OK as optimizations go. It was fairly easy to code, and it did get rid of an unproven assumption, but didn't really buy much in the long run.

#### Observation #2

Negative numbers are silly. Don't produce them as a new number in the sequence, as that's a ``stupid'' thing to do. Let's say the first negative number in the sequence is the number -42. Obviously -42 was constructed as the difference of two previous elements, so 42 could be constructed just as easily, and it could be used instead of -42 every time 42 was used. Of course, it would be even better if the code were written never to encounter -42.

New runtime: 387.34 seconds

Zero's are even sillier. Creating a silly answer doesn't give you any more ``power.'' Assume the shortest sequence contains a zero. Obviously, if the zero wasn't used in a later operation, it could be dropped from the sequence, resulting a shorter valid solution, so it must have been used. However adding zero and subtracting zero from some value does produces that same value, so any number produced in such a manner could also be dropped. The third alternative, subtract a value from zero results in a negative number, which as noted above, isn't helpful either. Thus, a zero will not be in the shortest sequence.

New runtime: 43.24 seconds

Evaluation: Two simple observations have reduced our runtime by a factor of 100. These are easy to code and give great improvements, the very embodiment of excellent optimizations.

**Observation #3**

In a single step, the largest possible generate-able integer is exactly double the maximum integer generated so far. Thus, if the maximum achieved so far times 2^(number of steps left until we reach the best found thus far) is less than the goal, there is no hope. Stop now.

Runtime: 0.15 seconds

Evaluation: This was moderately difficult to code, but bought a factor of 300 in runtime, so it was definitely worth it, assuming the contest coding time was available and the problem was running over the time limit.

**Test Set Update**

The most recent test set runs really quickly, so it's time to make the data set harder. On the new test data set of all powers from 1 to 300, the runtime is 93.21 seconds.

**Observation #4**

Why not do depth-first search with iterative deepening? The branching factor for this search is very large and grows quadratically as the depth increases, so the additional overhead is very small.

Runtime: 93.05 seconds

Evaluation: In this case, a poor optimization (by itself; things will improve later). It required quite a bit of code change, with a large chance of error, and yielded effectively nothing. This is pretty surprising, as DFSID generally helps immensely.

**Observation #5**

The most recent operation must use the next-to-last number created. If it didn't, why bother generating that number? (Note that this assumes the DFSID algorithm.)

Runtime: 7.47 seconds

**Observation #6**

Never duplicate a number in the list.

Runtime: 3.40 seconds

**Status Check**

Thus far, other than adding depth-first search with iterative deepening, only the ``Don't Do Anything Stupid'' rule has been used, and and the execution time has decreased by an estimated factor of 842,510. That's decent, but probably can be improved.

**Test set update**

New data set of 1 to 500. Runtime: 206.71 seconds

**Observation #7**

If a number is to be placed at the end of a sub-prefix chain that is searched completely finding no answer, then adding that same number later doesn't help. For example, if 1 2 4 8 7 ... doesn't work,there's no reason to try 1 2 4 8 16 7 ... later.

Runtime: 53.70 seconds

**Observation #8**

If the first number selected is i, and the largest number in the sequence is j, then if i + j is less than the minimum next number needed, there's no way to produce it using i.

Runtime: 44.52 seconds

**Observation #9**

If there a chain that produces x (where x is not the goal) in j steps, but there exists a sequence or length smaller than j which produces that same number, we can just replace that sequence with the shorter one, and obtain a ``better'' sequence, so any chain starting with this longer chain can't be optimal.

WRONG!

First counterexample: 10,127. This hypothesis yields a chain of 17 steps, when one of 16 steps exists.

This is the risk of optimizations: sometimes, they'll be based on incorrect facts. Make sure that you don't fall into this trap.

**Conclusion**

Eight optimizations yielded a total improvement factor of around 4 million. Additional changes can improve this beyond 44.52 seconds for 1 to 500. One good implementation takes 1.91 seconds, when limited to 640k of memory (0.85 seconds without). See how fast you can make your program.

USACO Gateway | Comment or Question

# Search Techniques

### Sample Problem: *n* Queens [Traditional]

Place *n* queens on an *n x n* chess board so that no queen is attacked by another queen.

### Depth First Search (DFS)

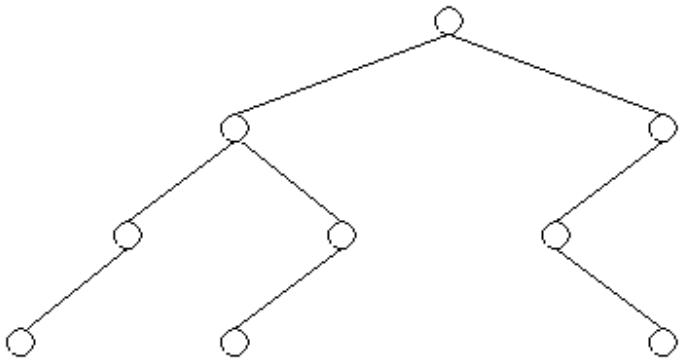
The most obvious solution to code is to add queens recursively to the board one by one, trying all possible queen placements. It is easy to exploit the fact that there must be exactly one queen in each column: at each step in the recursion, just choose where in the current column to put the queen.

```
1 search(col)
2     if filled all columns
3         print solution and exit

4   for each row
5       if board(row, col) is not attacked
6           place queen at (row, col)
7           search(col+1)
8           remove queen at (row, col)
```

Calling `search(0)` begins the search. This runs quickly, since there are relatively few choices at each step: once a few queens are on the board, the number of non-attacked squares goes down dramatically.

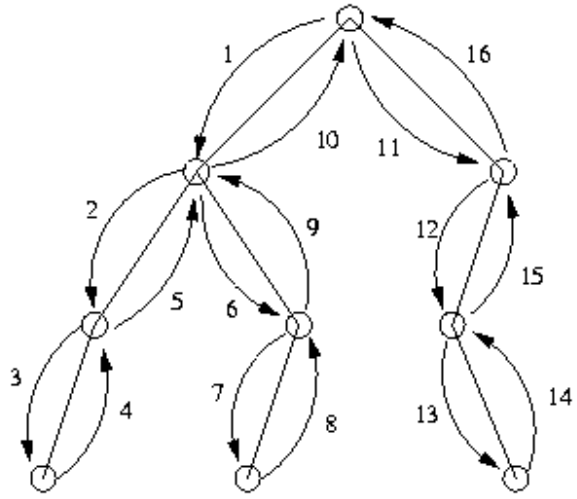
This is an example of *depth first search*, because the algorithm iterates down to the bottom of the search tree as quickly as possible: once *k* queens are placed on the board, boards with even more queens are examined before examining other possible boards with only *k* queens. This is okay but sometimes it is desirable to find the simplest solutions before trying more complex ones.

Depth first search checks each node in a search tree for some property. The search tree might look like this:



The algorithm searches the tree by going down as far as possible and then backtracking when necessary, making a sort of outline of the tree as the nodes are visited. Pictorially,

the tree is traversed in the following manner:




#### Complexity

Suppose there are *d* decisions that must be made. (In this case *d=n*, the number of columns we must fill.) Suppose further that there are C choices for each decision. (In this case *c=n* also, since any of the rows could potentially be chosen.) Then the entire search will take time proportional to $c^d$, i.e., an exponential amount of time. This scheme requires little space, though: since it only keeps track of as many decisions as there are to make, it requires only O(*d*) space.

### Sample Problem: Knight Cover [Traditional]

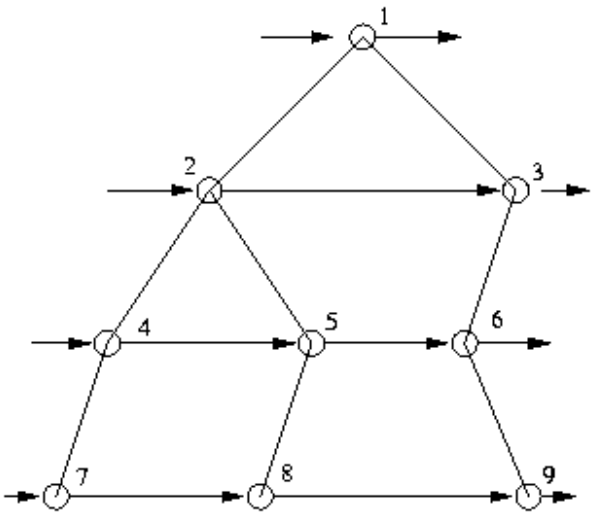
Place as few knights as possible on an *n x n* chess board so that every square is attacked. A knight is not considered to attack the square on which it sits.

#### Breadth First Search (BFS)

In this case, it is desirable to try all the solutions with only *k* knights before moving on to those with *k+1* knights. This is called **breadth first search**. The usual way to implement breadth first search is to use a queue of states:

```
1 process(state)
2     for each possible next state from this one
3         enqueue next state

4 search()
5     enqueue initial state
6     while !empty(queue)
7         state = get state from queue
8         process(state)
```

This is called breadth first search because it searches an entire row (the breadth) of the search tree before moving on to the next row. For the search tree used previously, breadth first search visits the nodes in this order:




It first visits the top node, then all the nodes at level 1, then all at level 2, and so on.

**Complexity**

Whereas depth first search required space proportional to the number of decisions (there were *n* columns to fill in the *n* queens problem, so it took O(*n*) space), breadth first search requires space exponential in the number of choices.

If there are *c* choices at each decision and *k* decisions have been made, then there are $c^k$ possible boards that will be in the queue for the next round. This difference is quite significant given the space restrictions of some programming environments.

[Some details on why $c^k$*: Consider the nodes in the recursion tree. The zeroeth level has 1 nodes. The first level has c nodes. The second level has* $c^2$ *nodes, etc. Thus, the total number of nodes on the k-th level is* $c^k$.]

**Depth First with Iterative Deepening (ID)**

An alternative to breadth first search is *iterative deepening*. Instead of a single breadth first search, run D depth first searches in succession, each search allowed to go one row deeper than the previous one. That is, the first search is allowed only to explore to row 1, the second to row 2, and so on. This ``simulates'' a breadth first search at a cost in time but a savings in space.

```
 1 truncated_dfsearch(hnextpos, depth)
 2     if board is covered
 3         print solution and exit

 4     if depth == 0
 5         return

 6     for i from nextpos to n*n
 7         put knight at i
 8         truncated_dfsearch(i+1, depth-1)
 9         remove knight at i

10 dfid_search
11     for depth = 0 to max_depth
12        truncated_dfsearch(0, depth)
```

**Complexity**

The space complexity of iterative deepening is just the space complexity of depth first search: O($n$). The time complexity, on the other hand, is more complex. Each truncated depth first search stopped at depth $k$ takes $c^k$ time. Then if $d$ is the maximum number of decisions, depth first iterative deepening takes $c^0 + c^1 + c^2 + ... + c^d$ time.

If $c = 2$, then this sum is $c^{d+1} - 1$, about twice the time that breadth first search would have taken. When $c$ is more than two (i.e., when there are many choices for each decision), the sum is even less: iterative deepening cannot take more than twice the time that breadth first search would have taken, assuming there are always at least two choices for each decision.

## Which to Use?

Once you've identified a problem as a search problem, it's important to choose the right type of search. Here are some things to think about.

#### In a Nutshell

| Search | Time | Space | When to use |
|---|---|---|---|
| DFS | O($c^k$) | O($k$) | Must search tree anyway, know the level the answers are on, or you aren't looking for the shallowest number. |
| BFS | O($c^d$) | O($c^d$) | Know answers are very near top of tree, or want shallowest answer. |
| DFS+ID | O($c^d$) | O($d$) | Want to do BFS, don't have enough space, and can spare the time. |

$d$ is the depth of the answer
$k$ is the depth searched
$d <= k$

Remember the ordering properties of each search. If the program needs to produce a list sorted shortest solution first (in terms of distance from the root node), use breadth first search or iterative deepening. For other orders, depth first search is the right strategy.

If there isn't enough time to search the entire tree, use the algorithm that is more likely to find the answer. If the answer is expected to be in one of the rows of nodes closest to the root, use breadth first search or iterative deepening. Conversely, if the answer is expected to be in the leaves, use the simpler depth first search.

Be sure to keep space constraints in mind. If memory is insufficient to maintain the queue for breadth first search but time is available, use iterative deepening.

## Sample Problems

#### Superprime Rib [USACO 1994 Final Round, adapted]

A number is called superprime if it is prime and every number obtained by chopping some number of digits from the right side of the decimal expansion is prime. For example, 233 is a superprime, because 233, 23, and 2 are all prime. Print a list of all the superprime numbers of length $n$, for $n <= 9$. The number 1 is not a prime.

For this problem, use depth first search, since all the answers are going to be at the $n$th level (the bottom level) of the search.

**Betsy's Tour [USACO 1995 Qualifying Round]**

A square township has been partitioned into $n^2$ square plots. The Farm is located in the upper left plot and the Market is located in the lower left plot. Betsy takes a tour of the township going from Farm to Market by walking through every plot exactly once. Write a program that will count how many unique tours Betsy can take in going from Farm to Market for any value of $n <= 6$.

Since the number of solutions is required, the entire tree must be searched, even if one solution is found quickly. So it doesn't matter from a time perspective whether DFS or BFS is used. Since DFS takes less space, it is the search of choice for this problem.

**Udder Travel [USACO 1995 Final Round; Piele]**

The Udder Travel cow transport company is based at farm A and owns one cow truck which it uses to pick up and deliver cows between seven farms A, B, C, D, E, F, and G. The (commutative) distances between farms are given by an array. Every morning, Udder Travel has to decide, given a set of cow moving orders, the order in which to pick up and deliver cows to minimize the total distance traveled. Here are the rules:

- The truck always starts from the headquarters at farm A and must return there when the day's deliveries are done.
- The truck can only carry one cow at a time.
- The orders are given as pairs of letters denoting where a cow is to be picked up followed by where the cow is to be delivered.

Your job is to write a program that, given any set of orders, determines the shortest route that takes care of all the deliveries, while starting and ending at farm A.

Since all possibilities must be tried in order to ensure the best one is found, the entire tree must be searched, which takes the same amount of time whether using DFS or BFS. Since DFS uses much less space and is conceptually easier to implement, use that.

**Desert Crossing [1992 IOI, adapted]**

A group of desert nomads is working together to try to get one of their group across the desert. Each nomad can carry a certain number of quarts of water, and each nomad drinks a certain amount of water per day, but the nomads can carry differing amounts of water, and require different amounts of water. Given the carrying capacity and drinking requirements of each nomad, find the minimum number of nomads required to get at least one nomad across the desert.

All the nomads must survive, so every nomad that starts out must either turn back at some point, carrying enough water to get back to the start or must reach the other side of the desert. However, if a nomad has surplus water when it is time to turn back, the water can be given to their friends, if their friends can carry it.

Analysis: This problem actually is two recursive problems: one recursing on the set of nomads to use, the other on when the nomads turn back. Depth-first search with iterative deepening works well here to determine the nomads required, trying first if any one can make it across by themselves, then seeing if two work together to get across, etc.

**Addition Chains**

An addition chain is a sequence of integers such that the first number is 1, and every subsequent number is the sum of some two (not necessarily unique) numbers that appear in the list before it. For example, 1 2 3 5 is such a chain, as 2 is 1+1, 3 is 2+1, and 5 is 2+3. Find the minimum length chain that ends with a given number.

Analysis: Depth-first search with iterative deepening works well here, as DFS has a tendency to first try 1 2 3 4 5 ... n, which is really bad and the queue grows too large very quickly for BFS.

USACO Gateway | Comment or Question

# Shortest Paths

**Sample Problem: Overfencing [Kolstad & Schrijvers, Spring 1999 USACO Open]**

Farmer John created a huge maze of fences in a field. He omitted two fence segments on the edges, thus creating two ``exits'' for the maze. The maze is a `perfect' maze; you can find a way out of the maze from any point inside it.

Given the layout of the maze, calculate the number of steps required to exit the maze from the `worst' point in the maze (the point that is `farther' from either exit when walking optimally to the closest exit).

Here's what one particular W=5, H=3 maze looks like:

```
+-+-+-+-+-+
|         |
+-+ +-+ + +
|     | | |
+ +-+-+ + +
| |     |
+-+ +-+-+-+
```

**The Abstraction**

Given:

- A directed graph with nonnegative weighted edges
- A path between two vertices of a graph is any sequence of adjacent edges joining them
- The shortest path between two vertices in a graph is the path which has minimal cost, where cost is the sum of the weights of edges in the path.

Problems often require only the cost of a shortest path not necessarily the path itself. This sample problem requires calculating only the costs of shortest paths between exits and interior points of the maze. Specifically, it requires the maximum of all of those various costs.

**Dijkstra's algorithm to find shortest paths in a weighted graph**

Given: lists of vertices, edges, and edge costs, this algorithm `visits' vertices in order of their distance from the source vertex.

- Start by setting the distance of all notes to infinity and the source's distance to 0.
- At each step, find the vertex u of minimum distance that hasn't been processed already. This vertex's distance is now frozen as the minimal cost of the shortest path to it from the source.
- Look at appending each neighbor v of vertex u to the shortest path to u. Check vertex v to see if this is a better path than the current known path to v. If so, update the best path information.

In determining the shortest path to a particular vertex, this algorithm determines all shorter paths from the source vertex as well since no more work is required to calculate *all* shortest paths from a single source to vertices in a graph.

Reference: Chapter 25 of [Cormen, Leiserson, Rivest]

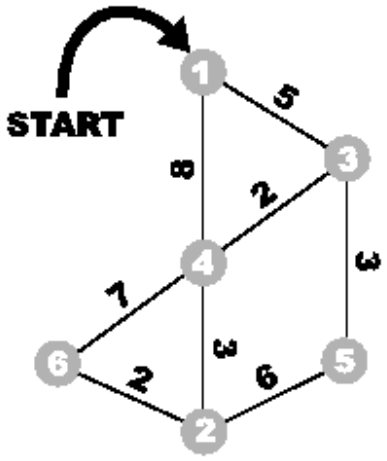
Pseudocode:

```
# distance(j) is distance from source vertex to vertex j
# parent(j) is the vertex that precedes vertex j in any shortest path
#                 (to reconstruct the path subsequently)

 1 For all nodes i
 2     distance(i) = infinity          # not reachable yet
 3     visited(i) = False
 4     parent(i) = nil # no path to vertex yet

 5 distance(source) = 0 # source -> source is start of all paths
 6 parent(source) = nil
 7   8 while (nodesvisited < graphsize)
 9     find unvisited vertex with min distance to source; call it vertex i
10     assert (distance(i) != infinity, "Graph is not connected")

11     visited(i) = True # mark vertex i as visited

    # update distances of neighbors of i
12     For all neighbors j of vertex i
13         if distance(i) + weight(i,j) < distance(j) then
14             distance(j) = distance(i) + weight(i,j)
15             parent(j) = i
```

Running time of this formulation is O($V^2$). You can obtain O($E$ log $V$) (where $E$ is the number of edges and $V$ is the number of vertices) by using a heap to determine the next vertex to visit, but this is considerably more complex to code and only appreciably faster on large, sparse graphs.

**Sample Algorithm Execution**

Consider the graph below, whose edge weights can be expressed two different ways:

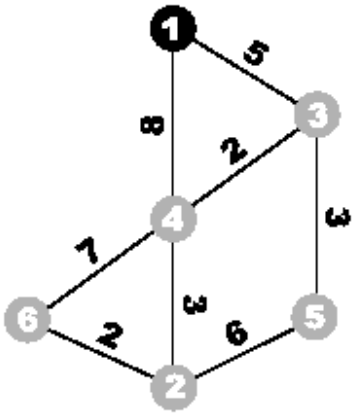



| Edge | Weight |
|---|---|
| (1, 3) | 5 |
| (1, 4) | 8 |
| (3, 4) | 2 |
| (3, 5) | 3 |
| (4, 2) | 3 |
| (4, 6) | 7 |
| (5, 2) | 6 |
| (2, 6) | 2 |

1 2 3 4 5 6
1 0 0 5 8 0 0
2 0 0 0 3 6 2
3 5 0 0 2 3 0
4 8 3 2 0 0 7
5 0 6 3 0 0 0
6 0 2 0 7 0 0

Here is the initial state of the program, both graphically and in a table:

| | | Distance to | |
|---|---|---|---|
| Node | Visited | Source | Parent |
| 1 | T | 0 | nil |
| 2 | F | infinity | nil |

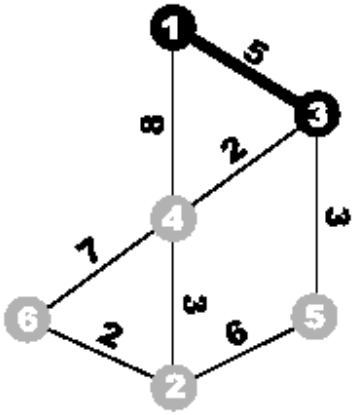



| 3 | F | infinity | nil |
|---|---|---|---|
| 4 | F | infinity | nil |
| 5 | F | infinity | nil |
| 6 | F | infinity | nil |

Updating the table, node 1's neighbors include nodes 3 and 4.

| | | Distance to | |
|---|---|---|---|
| Node | Visited | Source | Parent |
| 1 | T | 0 | nil |
| 2 | F | infinity | nil |
| 3 | F | **5** | **1** |
| 4 | F | **8** | **1** |
| 5 | F | infinity | nil |
| 6 | F | infinity | nil |

Node 3 is the closest unvisited node to the source node (smallest distance shown in column 3), so it is the next visited:

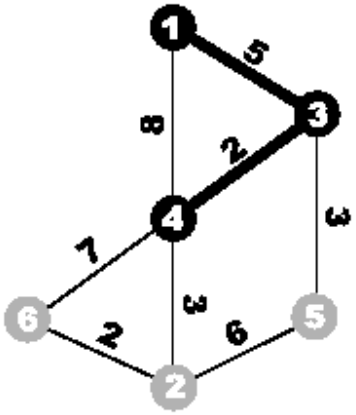



| | | Distance to | |
|---|---|---|---|
| Node | Visited | Source | Parent |
| 1 | T | 0 | nil |
| 2 | F | infinity | nil |
| 3 | **T** | 5 | 1 |
| 4 | F | 8 | 1 |
| 5 | F | infinity | nil |
| 6 | F | infinity | nil |

Node 3's neighbors are nodes 1, 4, and 5. Updating the unvisited neighbors yields:

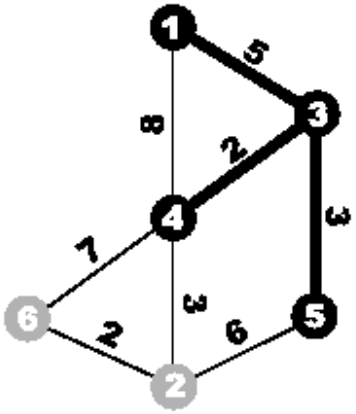
| | | Distance to | |
|---|---|---|---|
| Node | Visited | Source | Parent |
| 1 | T | 0 | nil |
| 2 | F | infinity | nil |
| 3 | T | 5 | 1 |
| 4 | F | **7** | **3** |
| 5 | F | **8** | **3** |
| 6 | F | infinity | nil |

Node 4 is the closest unvisited node to the source. Its neighbors are 1, 2, 3, and 6, of which only nodes 2 and 6 need be updated, since the others have already been visited:



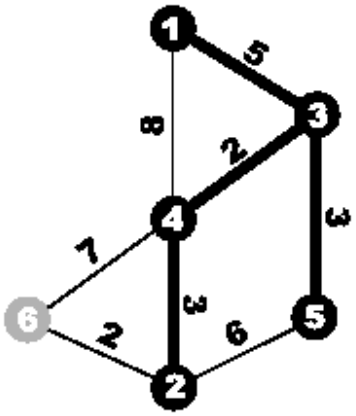

| | | Distance to | |
|---|---|---|---|
| Node | Visited | Source | Parent |
| 1 | T | 0 | nil |
| 2 | F | **10** | **4** |
| 3 | T | 5 | 1 |
| 4 | **T** | 7 | 3 |
| 5 | F | 8 | 3 |
| 6 | F | **14** | **4** |

Of the three remaining nodes (2, 5, and 6), node 5 is closest to the source and should be visited next. Its neighbors include nodes 3 and 2, of which only node 2 is unvisited. The distance to node 2 via node 5 is 14, which is longer than the already listed distance of 10 via node 4, so node 2 is not updated.



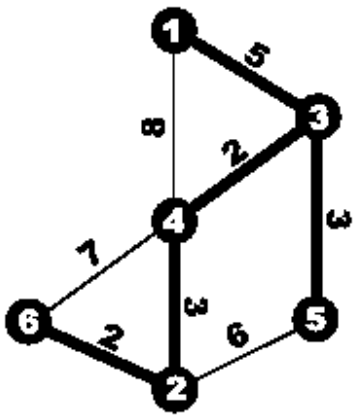

| | | Distance to | |
|---|---|---|---|
| Node | Visited | Source | Parent |
| 1 | T | 0 | nil |
| 2 | F | 10 | 4 |
| 3 | T | 5 | 1 |
| 4 | T | 7 | 3 |
| 5 | **T** | 8 | 3 |
| 6 | F | 14 | 4 |

The closest of the two remaining nodes is node 2, whose neighbors are nodes 4, 5, and 6, of which only node 6 is unvisited. Furthermore, node 6 is now closer, so its entry must be updated:




| | | Distance to | |
|---|---|---|---|
| Node | Visited | Source | Parent |
| 1 | T | 0 | nil |
| 2 | **T** | 10 | 4 |
| 3 | T | 5 | 1 |
| 4 | T | 7 | 3 |
| 5 | T | 8 | 3 |
| 6 | F | **12** | **2** |

Finally, only node 6 remains to be visited. All of its neighbors (indeed the entire graph) have now been visited:




| | | Distance to | |
|---|---|---|---|
| Node | Visited | Source | Parent |
| 1 | T | 0 | nil |
| 2 | T | 10 | 4 |
| 3 | T | 5 | 1 |
| 4 | T | 7 | 3 |
| 5 | T | 8 | 3 |
| 6 | **T** | 12 | 2 |

**Sample Problem: Package Delivery**

Given a set of locations, lengths of roads connecting them, and an ordered list of package dropoff locations. Find the length of the shortest route that visits each of the package dropoff locations in order.

Analysis: For each leg of the required path, run Dijkstra's algorithm to determine the shortest path connecting the two endpoints. If the number of legs in the journey exceeds N, instead of calculating each path, calculate the shortest path between all pairs of vertices, and then simply paste the shortest path for each leg of the journey together to get the entire journey.

**Extended Problem: All Pairs, Shortest Paths**

The extended problem is to determine a table *a*, where:
$a_{i,j}$ = length of shortest path between *i* and *j*, or infinity if *i* and *j* aren't connected.

This problem is usually solved as a subproblem of a larger problem, such as the package delivery problem.

Dijkstra's algorithm determines shortest paths for one source and all destinations in $O(N^2)$ time. We can run it for all *N* sources in $O(N^3)$ time.

If the paths do not need to be recreated, there's an even simpler solution that also runs in $O(N^3)$ time.

**The Floyd-Warshall Algorithm**

The Floyd-Warshall algorithm finds the length of the shortest paths between all pairs of vertices. It requires an adjacency matrix containing edge weights, the algorithm constructs optimal paths by piecing together optimal subpaths.

- Note that the single edge paths might not be optimal and this is okay.
- Start with all single edge paths. The distance between two vertices is the cost of the edge between them or infinity if there is no such edge.
- For each pair of vertices *u* and *v*, see if there is a vertex *w* such that the path from *u* to *v* through *w* is shorter than the current known path from to *u* to *v*. If so, update it.

- Miraculously, if ordered properly, the process requires only one iteration.
- For more information on why this works, consult Chapter 26 of [Cormen, Leiserson, Rivest].

Pseudocode:

```
# dist(i,j) is "best" distance so far from vertex i to vertex j

# Start with all single edge paths.
For i = 1 to n do
    For j = 1 to n do
        dist(i,j) = weight(i,j)

For k = 1 to n do # k is the `intermediate' vertex
    For i = 1 to n do
        For j = 1 to n do
            if (dist(i,k) + dist(k,j) < dist(i,j)) then # shorter path?
                dist(i,j) = dist(i,k) + dist(k,j)
```

This algorithm runs in $O(V^3)$ time. It requires the adjacency matrix form of the graph.

It's very easy to code and get right (only a few lines).

Even if the solution requires only the single source shortest path, this algorithm is recommended, provided the time and memory and are available (chances are, if the adjacency matrix fits in available memory, there is enough time).

## Problem Cues

If the problem wants an optimal path or the cost of a minimal route or journey, it is likely a shortest path problem. Even if a graph isn't obvious in a problem, if the problem wants the minimum cost of some process and there aren't many states, then it is usually easy to superimpose a graph on it. The big point here: shortest path = search for the minimal cost way of doing something.

## Extensions

If the graph is unweighted, the shortest path contains a minimal number of edges. A breadth first search (BFS) will solve the problem in this case, using a queue to visit nodes in order of their distance from the source. If there are many vertices but few edges, this runs much faster than Dijkstra's algorithm (see Amazing Barn in Sample Problems).

If negative weight edges are allowed, Dijkstra's algorithm breaks down. Fortunately, the Floyd-Warshall algorithm isn't affected so long as there are no negative cycles in the graph (if there is a negative cycle, it can be traversed arbitrarily many times to get ever `shorter' paths). So, graphs must be checked for them before executing a shortest path algorithm.

It is possible to add additional conditions to the definition of shortest path (for example, in the event of a tie, the path with fewer edges is shorter). So long as the distance function can be augmented along with the comparison function, the problem remains the same. In the example above, the distance function contains two values: weight and edge count. Both values would be compared if necessary.

## Sample Problems

### Graph diameter

Given: an undirected, unweighted, connected graph. Find two vertices which are the farthest apart.

Analysis: Find the length of shortest paths for all pairs and vertices, and calculate the maximum of these.

**Knight moves**

Given: Two squares on an NxN chessboard. Determine the shortest sequence of knight moves from one square to the other.

Analysis: Let the chessboard be a graph with 64 vertices. Each vertex has at most 8 edges, representing squares 1 knight move away.

**Amazing Barn (abridged) [USACO Competition Round 1996]**

Consider a very strange barn that consists of N stalls (N < 2500). Each stall has an ID number. From each stall you can reach 4 other stalls, but you can't necessarily come back the way you came.

Given the number of stalls and a formula for adjacent stalls, find any of the `most central' stalls. A stall is `most central' if it is among the stalls that yields the lowest average distance to other stalls using best paths.

Analysis: Compute all shortest paths from each vertex to determine its average distance. Any $O(N^3)$ algorithm for computing all-pairs shortest paths would be prohibitively expensive here since N=2500. However, there are very few edges (4 per vertex), making a BFS with queue ideal. A BFS runs in O(E) time, so to compute shortest paths for all sources takes O(VE) time - about:
$2500 \times 10{,}000 = 2.5 \times 10^6$ things, much more reasonable than $2500^3 = 1.56 \times 10^{10}$

**Railroad Routing (abridged) [USACO Training Camp 1997, Contest 1]**

Farmer John has decided to connect his dairy cows directly to the town pasteurizing plant by constructing his own personal railroad. Farmer John's land is laid out as a grid of one kilometer squares specified as row and column.

The normal cost for laying a kilometer of track is $100. Track that must gain or lose elevation between squares is charged a per-kilometer cost of $100 + $3 x meters_of_change_in_elevation. If the track's direction changes 45 degrees within a square, costs rise an extra $25; a 90 degree turn costs $40. All other turns are not allowed.

Given the topographic map, and the location of both John's farm and the plan, calculate the cost of the cheapest track layout.

Analysis: This is almost a standard shortest path problem, with grid squares as vertices and rails as edges, except that the direction a square is entered limits the ways you can exit that square. The problem: it is not possible to specify which edges exist in advance (since the path matters).

The solution: create eight vertices for each square, one for each direction you can enter that square. Now you can determine all of the edges in advance and solve the problem

as a shortest path problem.

USACO Gateway | Comment or Question