

1. Data Structures

1.1 Dictionaries

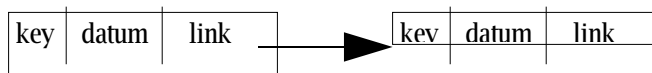
A dictionary is a set of items with a key attached to each item. Using the key, we can access, insert or delete the item quickly. Examples are a symbol table in a compiler, a database of student academic records, shortest distances maintained by an algorithm dealing with graphs. Thus a dictionary can be a part of system program, or directly used for practical purposes, or even can be a part of another algorithm.

The keys must be from a linearly ordered set.. That is we can answer the question $key1 < key2$, $key1 > key2$, or $key1 = key2$ for any two keys $key1$ and $key2$. We often omit the data in each item from an algorithmic point of view and only deals with keys.

A linearly ordered list is not very efficient to maintain a dictionary. An example of linked list structure is given.

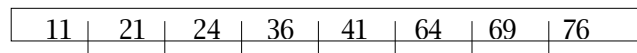


A detailed structure of each cell consists of three fields as show below.



If we have n keys, we need $O(n)$ time to access an item holding the key in the worst case. Operations like insert and delete will be easy in the linked structure since we can modify the linking easily.

If we keep the items in a one-dimensional array (or simply an array), it is easy to find a key by binary search, but insert and delete will be difficult because we have to shift the elements to the right or to the left all the way, causing $O(n)$ time. See below.



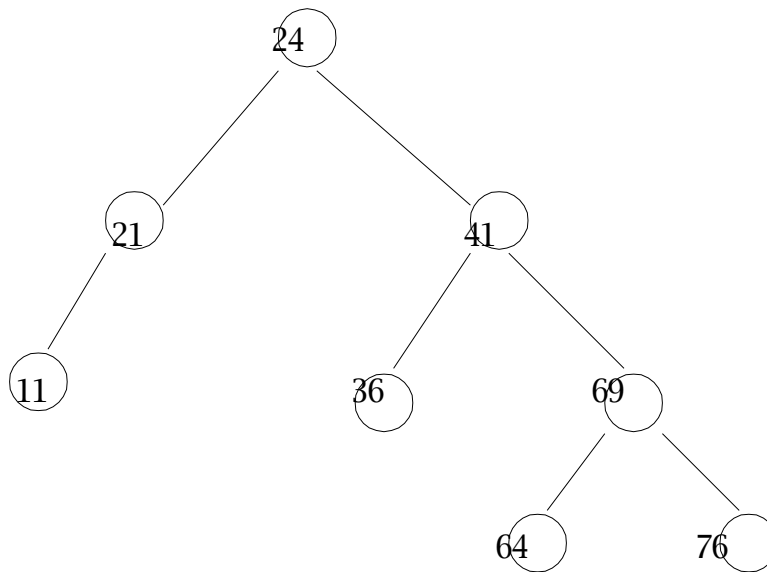
In the linked structure, the keys need not be sorted, whereas in the array version, they must be sorted for finding a key efficiently by binary search. In binary search, we look at the mid point of the array. If the key we are looking for is smaller, we go to the left half,

if greater we got to the right half, and if equal we are done. Even if the key does not exist, the search will finish in $O(\log n)$ time.

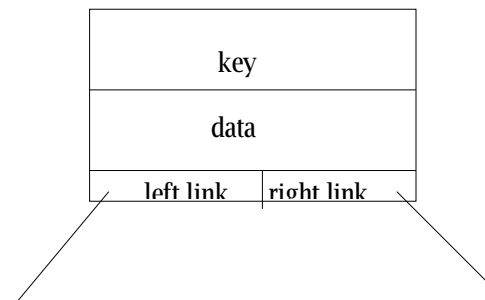
In dictionaries, we implement various operations like “create”, “find”, “insert”, “delete”, “min”, etc.

1.2 Binary Search Tree

Let us organise a dictionary in a binary tree in such a way that for any particular node the keys in the left subtree are smaller than or equal to that of the node and those in the right subtree are greater than or equal to that of the node. An example is given.



This kind of tree is called a binary search tree. A detailed structure of each node consists of four fields as shown below.

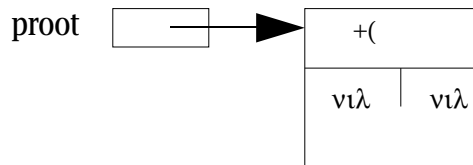


to the left child

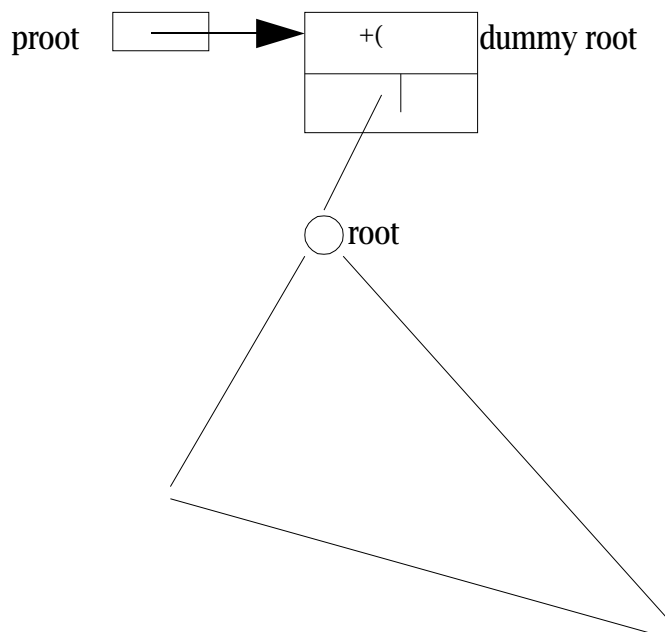
to the right child

There are many ways we can organise the set of keys {11, 21, 24, 36, 41, 64, 69, 76} in a binary search tree. The shape of the tree depends on how the keys are inserted into the tree. We omit the data field in the following.

The operation create(root) returns the following structure. This stands for an empty tree. We put an a node with the key value of infinity for programming ease.



After we insert several items, we have the following structure.



The number of items in the tree is maintained by the variable n. The “find” operation is implemented in the following with key x to be found.

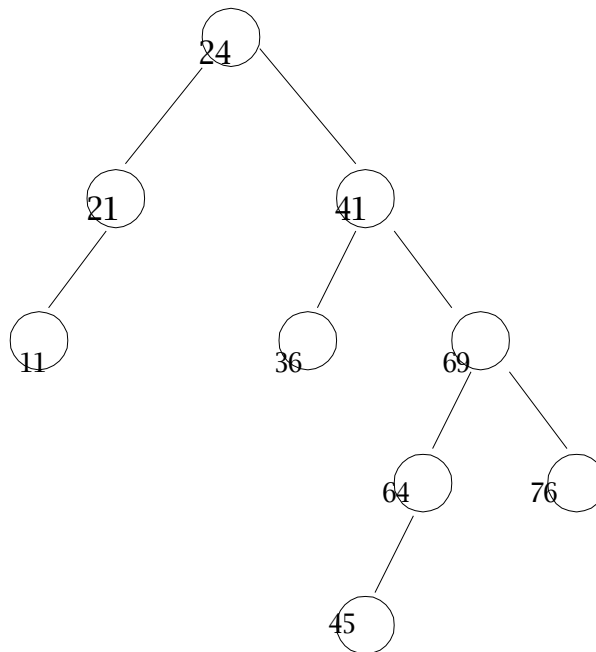
1. **begin** p:=proot; found:=false;
2. **repeat** q:=p;
3. **if** x < p^.key **then** p:=q^.left

4. **else** if $x > p.^{key}$ **then** $p:=q.^{right}$
5. **else** $\{x = p.^{key}\}$ $found:=true$
6. **until** $found$ **or** $(p = nil)$
7. **end.**
8. $\{$ If $found$ is true, key x is found at position pointed to by p
9. If $found$ is false, key x is not found and it should be inserted at the position pointed to by q $\}$

The operation “insert(x)” is implemented with the help of “find” in the following.

1. **begin** find(x); $\{found$ should be false $\}$
2. $n:=n+1$;
3. new(w);
4. $w.^{key}:=x$; $w.^{left}:=nil$; $w.^{right}:=nil$;
5. **if** $x < q.^{key}$ **then** $q.^{left}:=w$ **else** $q.^{right}:=w$
6. **end.**

Example. If we insert 45 into the previous tree, we have the following.



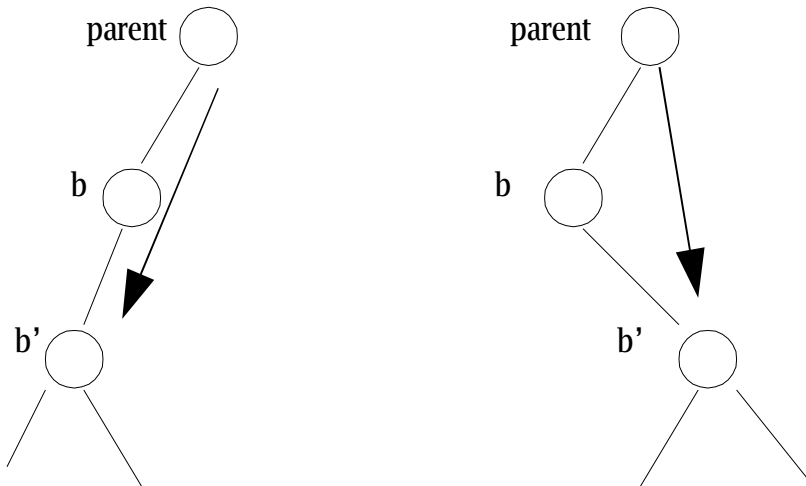
The operation “min” which is to find the minimum key in the tree is implemented in the following.

1. **begin** $p:=root.^{left}$;

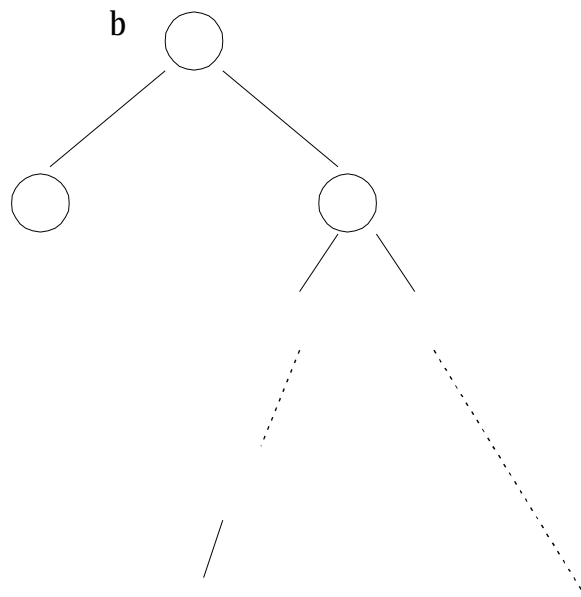
2. **while** $p^{left} = nil$ **do** $p := p^{left}$;
3. $min := p^{key}$
4. **end**
5. {The node with the minimum key min is pointed to by p }.

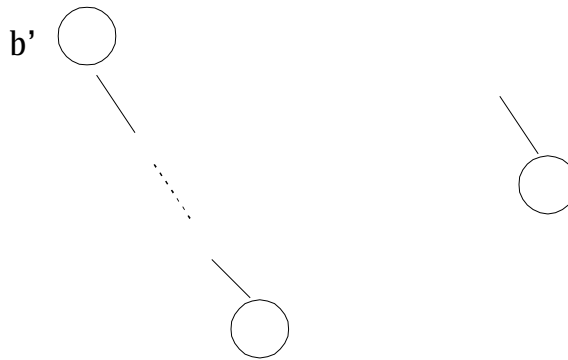
The “delete” operation is implemented with “min”. We delete the node b pointed to by p whose key is x .

If b has only one child, we can just connect the parent and the child. See the following.



If b has two children, we find the minimum b' in the right subtree of b , and move b' to b as shown below.

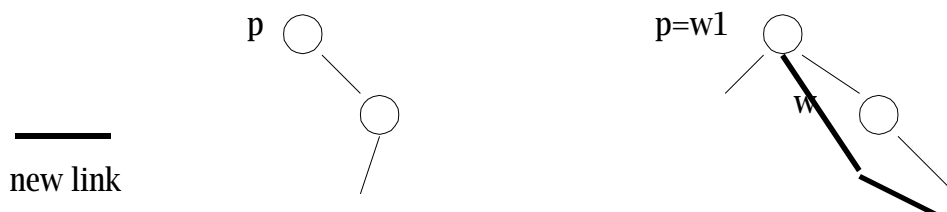


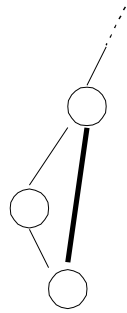


The operation delete is now described in the following.

1. {node at p whose key is x is to be deleted}
2. **begin** n:=n-1;
3. **if** p[^].right ≠ nil **and** p[^].left ≠ nil **then**
4. **begin** w1:=p; w:=p[^].right;
5. **while** w[^].left ≠ nil **do**
6. **begin** w1:=w; w:=w[^].left **end**;
7. **if** w1 ≠ p **then** w1[^].left := w[^].right
8. **else** w1[^].right := w[^].right;
9. p[^].key := w[^].key
10. {and possibly p[^].data := w[^].data}
11. **end**
12. **else begin**
13. **if** p[^].left ≠ nil **then** w:= p[^].left
14. **else** {p[^].right ≠ nil} w:= p[^].right;
15. w1:=proot;
16. **repeat** w2:=w1;
17. **if** x < w1[^].key **then** w1:= w1[^].left **else** w1:= w1[^].right
18. **until** w1 = p;
19. **if** w2[^].left = p **then** w2[^].left := w **else** w2[^].right := w
20. **end.**

The situations at lines 7 and 8 are depicted below.





at line 7



at line 8

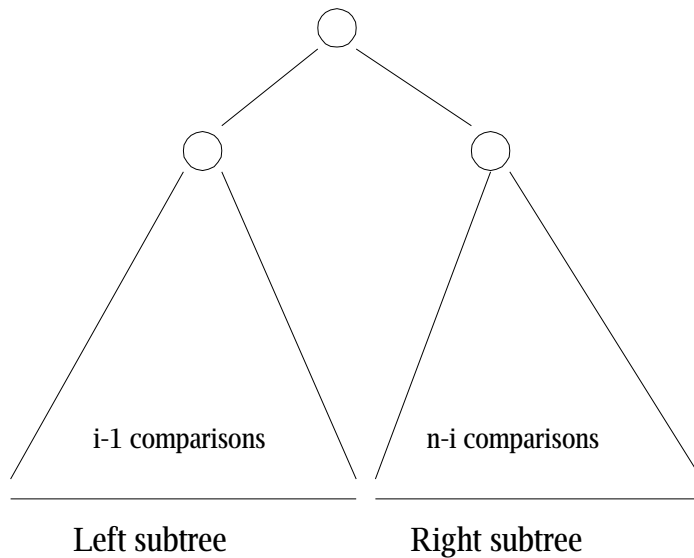
Analysis of n successive insertions in binary search tree

Let us insert n numbers (keys) into a binary search tree in random order. Let $T(n)$ be the number of comparisons needed. Then we have the following recurrence relation.

$$T(1) = 0$$

$$T(n) = n-1 + (1/n) \sum_{i=1}^{n-1} (T(i-1) + T(n-i)).$$

This is seen from the following figure.



Suppose the key at the root is the i -th smallest in the entire tree. The elements in the left subtree will consume $i-1+T(i-1)$ comparisons and those in the right subtree will take $n-i+T(n-1)$ comparisons. This event will take place with probability $1/n$.

Solution of $T(n)$

$$T(n) = n-1 + (2/n)\sum_{[i=0 \text{ to } n-1]} T(i) \quad , \text{ or } nT(n) = n(n-1) + 2\sum_{[i=0 \text{ to } n-1]} T(i)$$

$$T(n-1) = n-2+(2/(n-1))\sum_{[i=0 \text{ to } n-2]} T(i), \text{ or } (n-1)T(n-1) = (n-1)(n-2)+2\sum_{[i=0 \text{ to } n-2]} T(i)$$

Subtracting the second equation from the first yields

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= n(n-1) - (n-1)(n-2) + 2T(n-1) \\ &= 2n - 2 + 2T(n-1) \end{aligned}$$

That is,

$$nT(n) = 2(n-1) + (n+1)T(n-1).$$

Dividing both sides by $n(n+1)$, we have

$$T(n) / (n+1) = 2(n-1)/(n(n+1)) + T(n-1)/n$$

$$\text{i.e., } T(n) / (n+1) = 4/(n+1) - 2/n + T(n-1)/n.$$

Repeating this process yields

$$\begin{aligned} T(n) / (n+1) &= 4/(n+1) - 2/n + 4/n - 2/(n-1) + \dots + 4/3 - 2/2 + T(1) \\ &= 4/(n+1) + 2(1/n + \dots + 1/2 + 1/1) - 4 \\ &= 2\sum_{[i=1 \text{ to } n]} 1/i + 4/(n+1) - 4 \end{aligned}$$

That is,

$$\begin{aligned} T(n) &= 2(n+1) \sum_{[i=1 \text{ to } n]} 1/i - 4n \\ &= 2(n+1) H_n - 4n, \end{aligned}$$

where H_n is the n -th harmonic number, which is approximately given by

$$H_n \cong \log_e n - \gamma \quad (n \rightarrow \infty), \quad \gamma = 0.5572 \text{ Euler's constant}$$

That is,

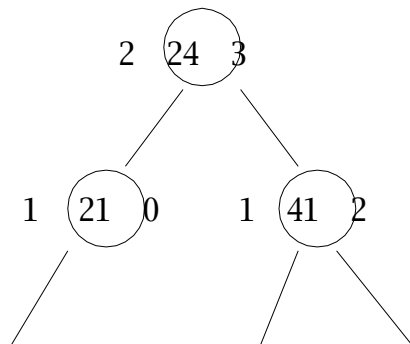
$$T(n) \cong 1.39 \log_2 n - 4n \quad (n \rightarrow \infty)$$

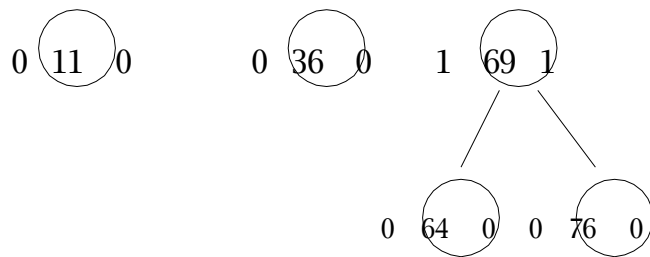
1.3 AVL Trees

Roughly speaking, we can perform various operations such as insert, delete, find, etc. on a binary search tree in $O(\log n)$ time on average when keys are random. In the worst case, however, it takes $O(n^2)$ time to insert the keys in increasing order. There are many ways to maintain the balance of the tree. One such is an AVL tree, invented by Adel'son-Velskii and Landis in 1962. An AVL tree is a binary search tree satisfying the following condition at each node.

The length of the longest path to leaves in the left subtree and the length of the longest path to leaves in the right subtree differ by at most one.

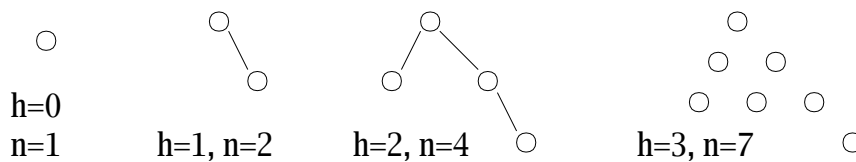
Example. The former example is an AVL tree. The longest lengths are attached to each node.



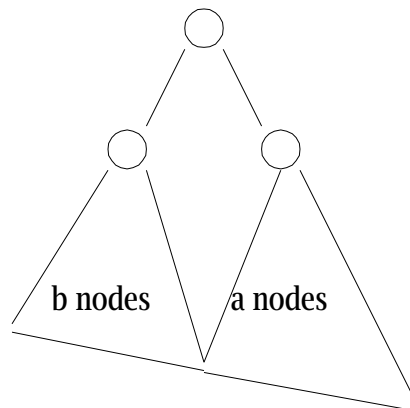


Analysis of AVL trees

Let the longest path from the root to the leaves of an AVL tree T with n nodes be $h_T(n)$, which is called the height of T . Let $h(n)$ be the maximum of $h_T(n)$ among AVL trees T with n nodes. We call this tree the tallest AVL tree, or simply tallest tree. Small examples are given below.



We have the following recurrence for the general situation.



There are n nodes
in this tree

$$h(1) = 0$$

$$h(2) = 1$$

$$h(a) = h(b) + 1$$

$$h(n) = h(a) + 1$$

$$n = a + b + 1$$

$$H(0) = 1$$

$$H(1) = 2$$

$$H(h) = H(h-1) + H(h-2) + 1$$

Let $H(h)$ be the inverse function of h , that is, $H(h(n)) = n$. $H(h)$ is the smallest number of nodes of AVL trees of height h . The solution of $H(h)$ is given by

$$H(h) = (\alpha^{h+3} - \beta^{h+3}) / \sqrt{5} - 1$$

$$\alpha = (1 + \sqrt{5}) / 2, \beta = (1 - \sqrt{5}) / 2.$$

From this we have

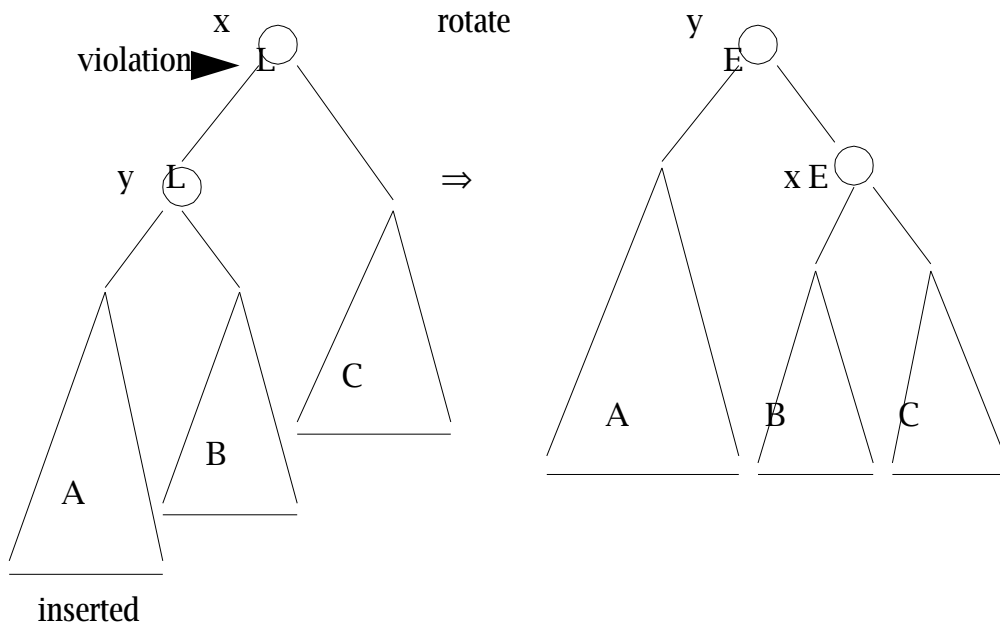
$h \cong \log_{\alpha} n = 1.45 \log_2 n$. That is, the height of AVL trees are not worse than completely balanced trees by 45%.

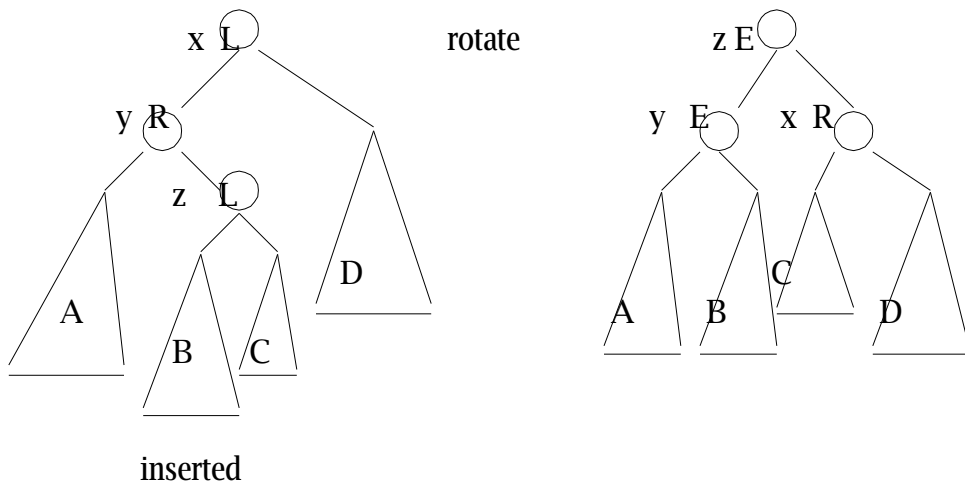
Rotation of AVL trees

As insert an item into an AVL tree using the insertion procedure for an binary search tree, we may violate the condition of balancing. Let markers L, E, R denote the situations at each node as follows:

- L : left subtree is higher
- E : they have equal heights
- R : right subtree is higher.

After we insert a new item at a leaf, we trace the path back to the root and rotate the tree if necessary. We have the following two typical situations. Let the marker at x is old and all others are new.

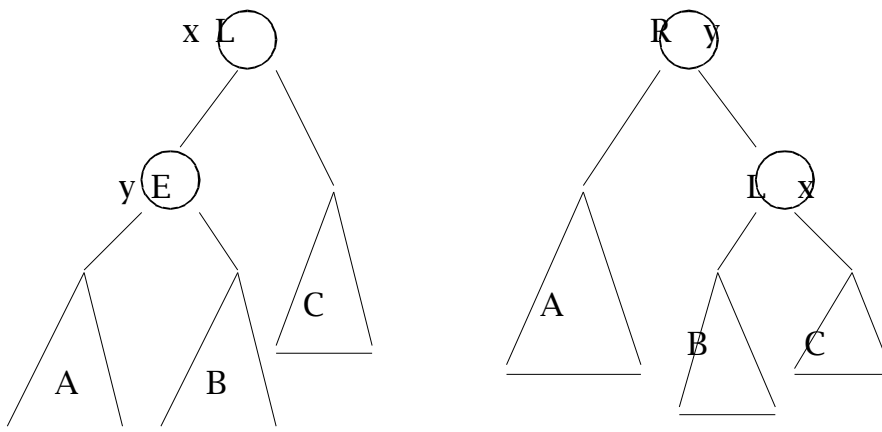




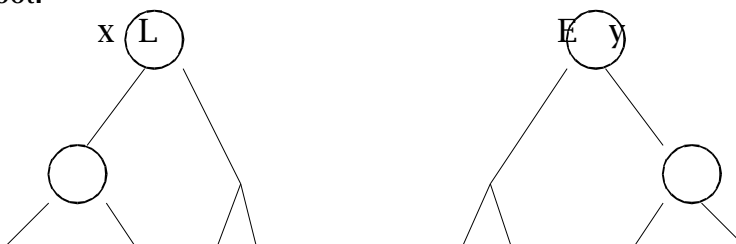
Exercise. Invent a rotation mechanism for deletion.

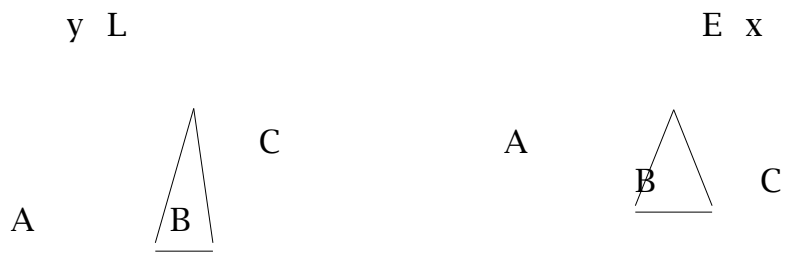
Deletion in an AVL Tree

Case 1. Deletion occurred in C. After rotation, we can stop.

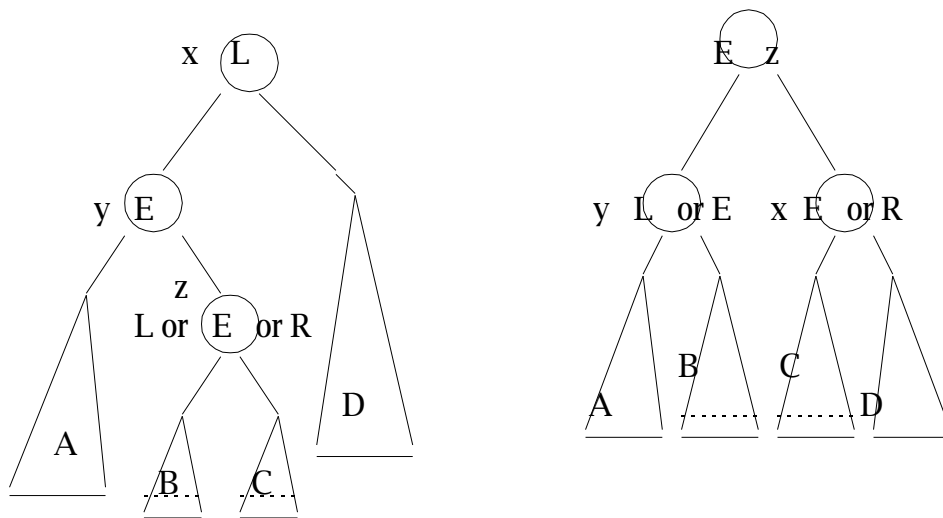


Case 2. Deletion occurred in C. We lost the height by one, and need to keep adjustment towards root.





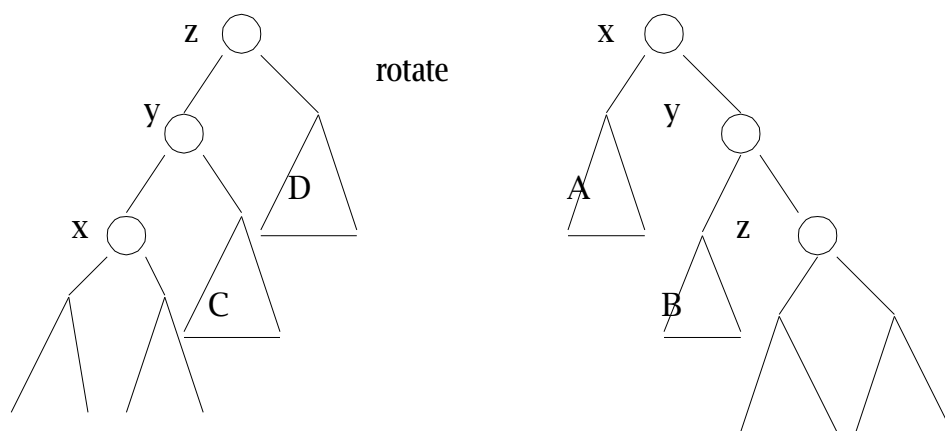
Case 3. Deletion occurred in D. We lost the height by one, and need to keep adjustment towards root.



1.4 Splay Trees

A splay tree is a binary search tree on which splay operations are defined. When we access a key x in the tree, we perform the following splay operations. Symmetric cases are omitted.

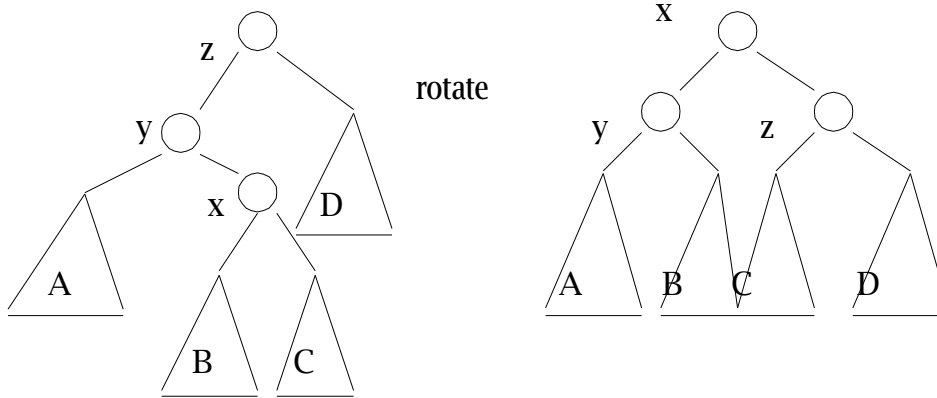
Case 1



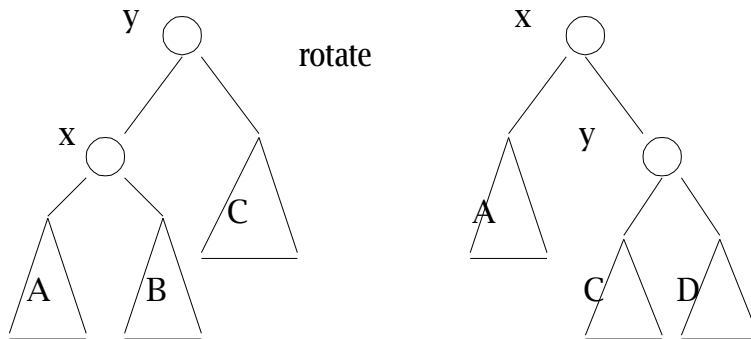
A B

 C D

Case 2



Case 3

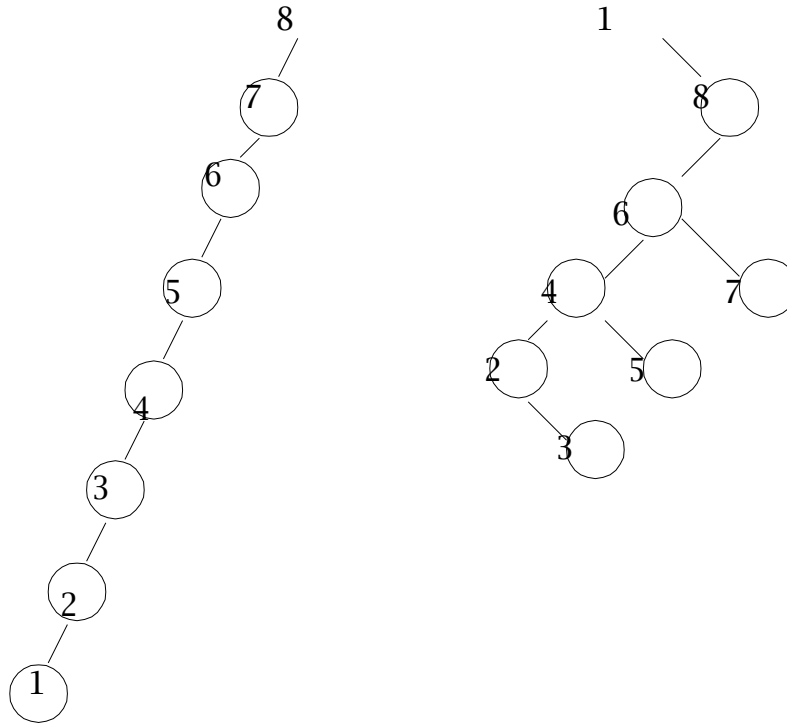


We continue case 1 or case 2 towards the root. At the root y, we perform case 3. This whole process is called splaying. The three operations find, insert and delete are described as follows:

- find(x) : splay originating at x
- insert(x): splay originating at x
- delete(x): splay originating at the root of x.

The splay operation contributes to changing the shape of the tree to a more balanced shape. Although each operation takes $O(n)$ time in the worst case, the splay operation will contribute to the saving of the computing time in the future. An example is shown below.





Definition. Amortised time is defined in the following way.

$$\text{amortised time} = \text{actual time} - \text{saving}$$

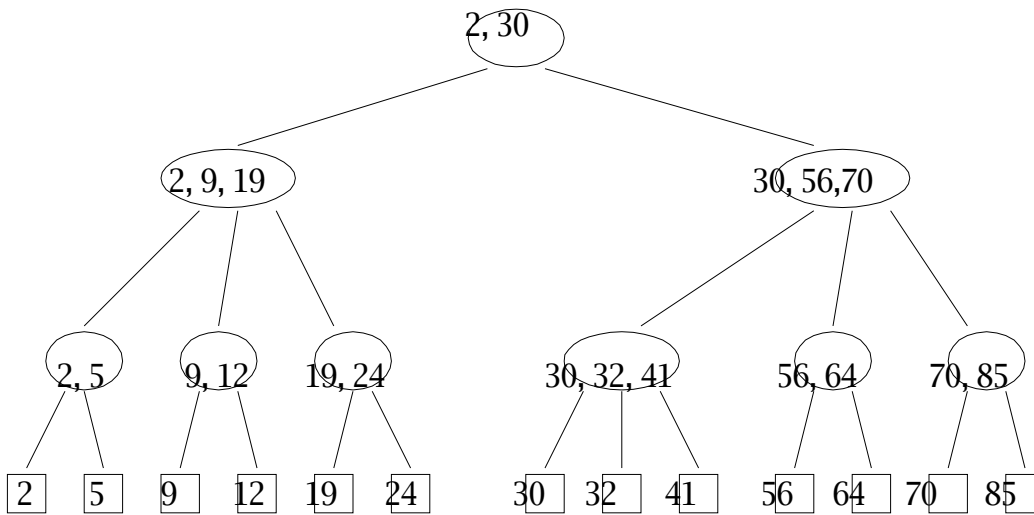
The concept of saving is defined by various means in various algorithms. In splay trees, amortised time is shown to be $O(\log n)$. The concept of amortised analysis is important when we perform many operations. It is known that m splay operations take $O((m+n)\log n)$ actual time through the amortised analysis..

1.5 B-trees

The B-tree was invented by Bayer and McCreight in 1972. It maintains balance, i.e., the paths from the root to leaves are all equal. There are several variations of B-trees. Here we describe the 2-3 tree in which we maintain items at leaves and internal nodes play the role of guiding the search.

An internal node of a 2-3 tree has 2 or 3 children and maintain 2 or 3 keys.

Example



Internal nodes are round and external nodes (leaves) are square. a node that has i children is called an i -node. An i -node has i keys, key_1 , key_2 and key_3 (if any). key_i is the smallest key of the leaves of the i -th subtree of the i -node.

The operation $find(x)$ is described as follows:

1. Go to the root and make it the current node.
2. Let key_1 , key_2 and key_3 (if any) be the keys of the current node.
3. If $x < key_2$, go to the first child.
4. If $key_2 \leq x < key_3$ (if any), go to the second child.
5. If $key_3 \leq x$, go to the third child (if any).
6. Repeat from 2 until current node is a leaf.
7. If $x = key$ of the leaf, report "found".
8. If $x < key$, x is to be inserted to the left.
9. If $x > key$, x is to be inserted to the right.

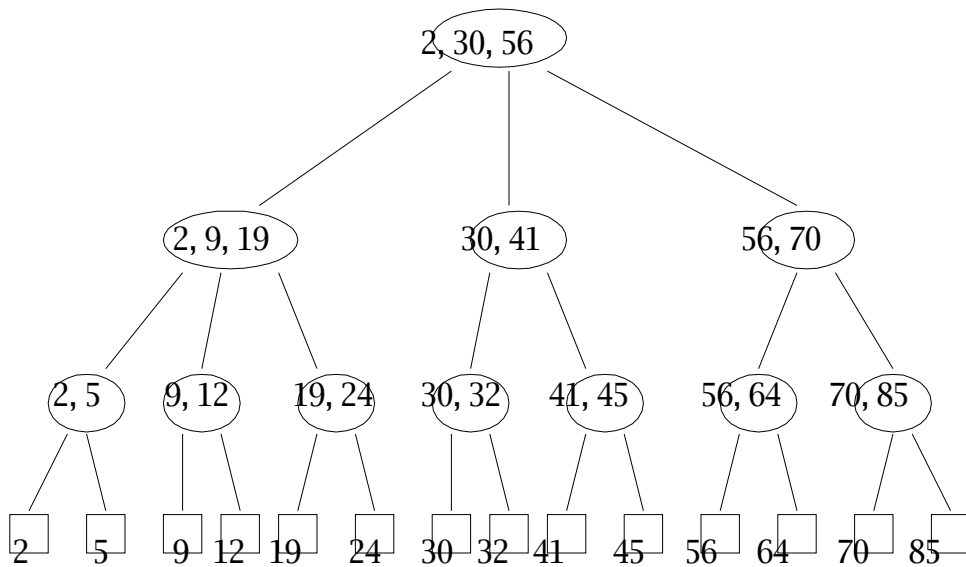
The height of the tree is $O(\log n)$ and $find(x)$ takes $O(\log n)$ time. Insertion and deletion cause reorganisation of the tree.

If we insert an item at the bottom and the parent had two children originally, there will be no reorganisation, although the keys along the path taken by $find(x)$ will be updated.

Example. If we insert 28, it is inserted to the right of 24 and the parent will have $key_3 = 28$.

If the parent gets four children after insertion, it is split into two and reorganisation starts. If the parent of parent gets four as the results of the splitting, this reorganisation propagates towards the root. The keys over the path will be updated. Key1 is redundant for find(x), but convenient for updating keys held by internal nodes. It is obvious that the whole reorganisation takes $O(\log n)$ time.

Example. If we insert 45 into the previous tree, we have the following.



Exercise. Invent a reorganisation mechanism for deletion.

1.6 Heap ordered trees (or priority queues)

A heap ordered tree is a tree satisfying the following condition.

“The key of a node is not greater than that of each child if any”

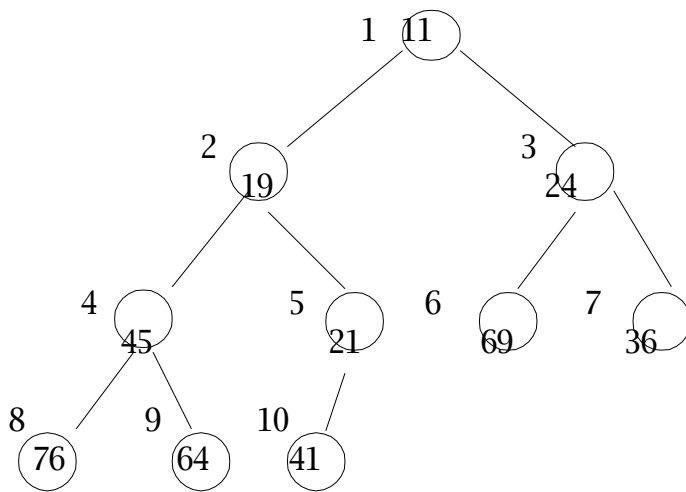
In a heap ordered tree, we can not implement “find” efficiently, taking $O(n)$ time when the size of the tree is n . We implement the following operations efficiently on a heap ordered tree. Note that “find” is missing.

- min
- update_key (decrease_key, increase_key)
- delete
- insert
- meld

1.6.1 The heap

The heap is a heap ordered tree of the complete binary tree shape. In general, a heap ordered tree needs a linked structure which requires some overhead time to traverse, whereas a heap can be implemented in an array on which heap operations efficiently work.

Example



The labels attached to nodes represent the array indices when implemented on an array

11	19	24	45	21	69	36	76	64	41
1	2	3	4	5	6	7	8	9	10

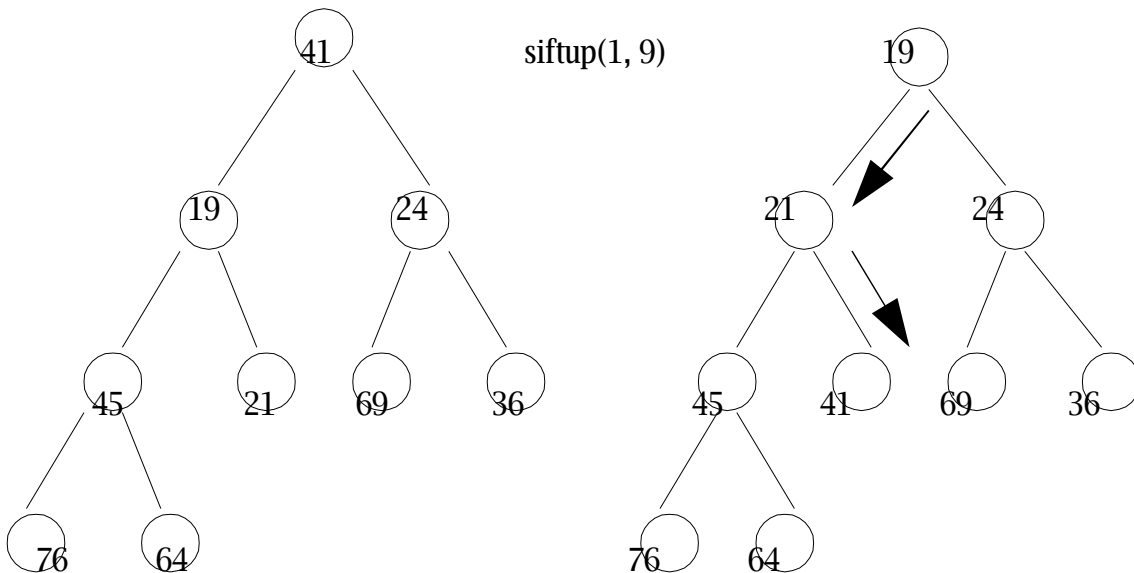
A complete binary tree occupies the first n elements in an array. If a node i (indicated by index i) has one or two children the left child is at $2i$ and the right child (if any) is at $2i+1$. The parent of node j ($j>1$) is at $j \div 2$. These operations can be implemented by shift operations in machine code and thus fast.

Suppose the given heap is contained in $a[1 .. n]$. The procedure $\text{siftup}(p, q)$ adjusts the subtree rooted at p so that the whole tree can be recovered to be a heap when only the key at p may be greater than those of its children and thus violates the heap condition. The parameter q indicates the end of the subtree.

1. **procedure** $\text{siftup}(p, q)$;
2. **begin**
3. $y:=a[p]$; $j:=p$; $k:=2*p$;
4. **while** $k \leq q$ **do begin**
5. $z:=a[k]$;
6. **if** $k < q$ **then if** $z > a[k+1]$ **then begin** $k:=k+1$; $z:=a[k]$ **end**;
7. **if** $y \leq z$ **then goto** 10;
8. $a[j]:=z$; $j:=k$; $k:=2*j$
9. **end**;
10. $a[j]:=y$
11. **end**;

The item with a smaller key is repeatedly compared with y and go up the tree each one level until y finds its position and settles at line 10. $O(\log n)$ time.

Example. Pick up the key 41 at node 10 and put it at node 1. The key 41 comes down and we have the following.



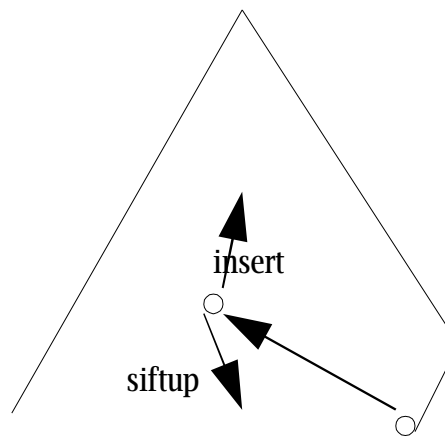
The procedure $\text{insert}(x)$ is described below. $O(\log n)$ time.

1. **procedure** insert(x);
2. **begin**
3. n:=n+1;
4. i:=n;
5. **while** i >= 2 **do begin**
6. j := i **div** 2;
7. y := a[j];
8. **if** x >= y **then go to** 10;
9. a[i] := y; i:=j
10. **end;**
11. 10: a[i] := x
12. **end;**

The key x is put at the end of heap and if the key value is small, we go up the path toward the root to find a suitable place for x.

The delete operation is a combination of insert and siftup operations. $O(\log n)$ time.

1. {x=a[p] is to be deleted}
2. **begin**
3. n:=n-1;
4. **if** p <= n **then**
5. **if** a[p] <= a[n+1] **then begin**
6. a[p] := a[n+1];
7. siftup(p, n)
8. **end else begin**
9. m:=n; n:=p-1;
10. insert(a[m+1]); n:=m
11. **end**
12. **end.**

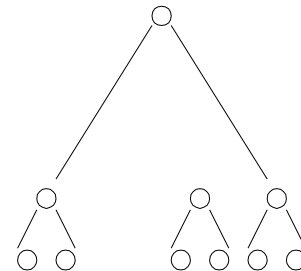


The “min” operation is easy. The minimum is at the root. $O(1)$ time. The operation decrease_key(p) is similar to insert and increase_key(p) is similar to siftup, where p is the node index where the key value is changed. These take $O(\log n)$ time.

The “meld” operation which melds two priority queues is not easy for the heap. This is easy for the binomial heap which will be described in the next section.

The operation build_heap is to create a heap given in the following.

1. **procedure** build_heap;
2. **begin**
3. **for** i:=n **div** 2 **downto** 1 **do** siftup(i, n)
4. **end.**



This procedure create a heap in a bottom-up manner.

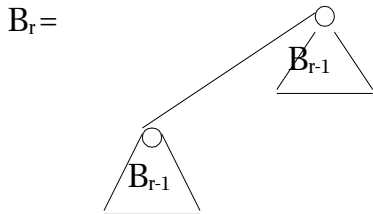
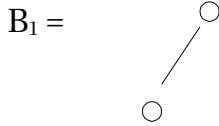
The computing time can be shown to be $O(n)$.



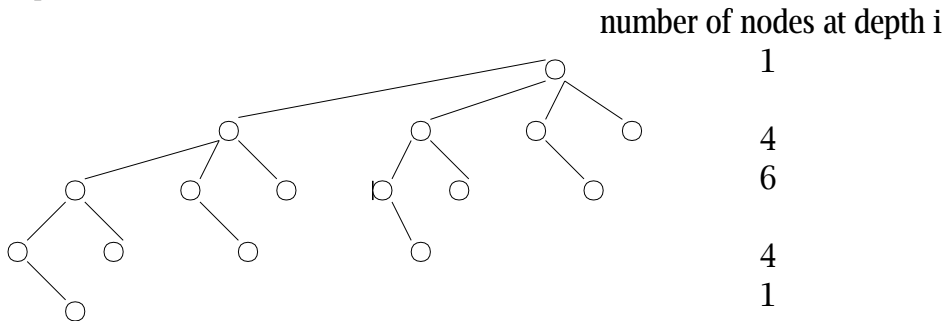
1.6.2 Binomial Queues

A binomial tree B_r is recursively defined as follows:

$$B_0 = \circ$$



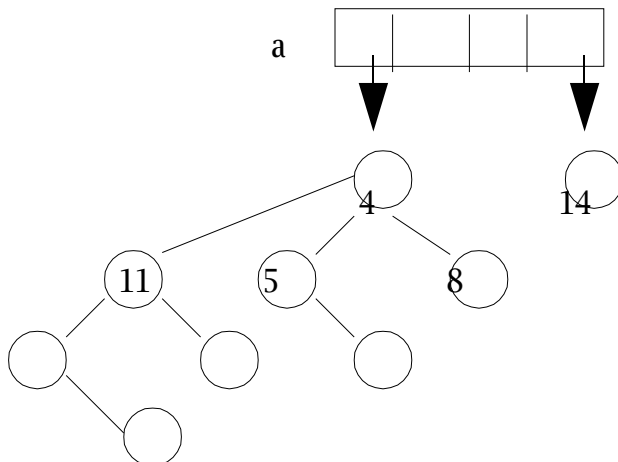
Example B_4



At depth i , there are $C(r, i)$ nodes, where $C(r, i) = r!/i!$ is a binomial coefficient, hence the name.

A binomial queue is a forest of binomial trees. The root of each tree in the forest is pointed to by an array element.

Example. $n=9$ in a binary form is $(1001)_2$. Then the binomial queue consists of B_3 and B_0 as shown below.

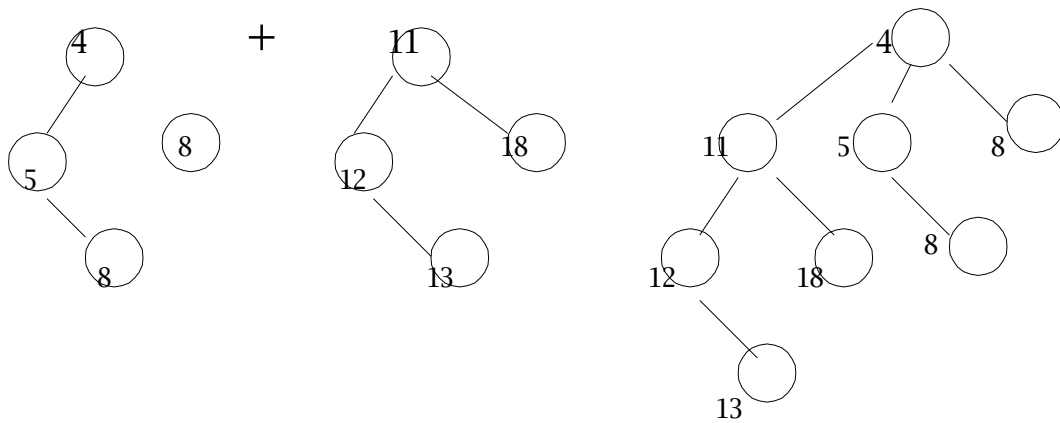


12 18 8

13

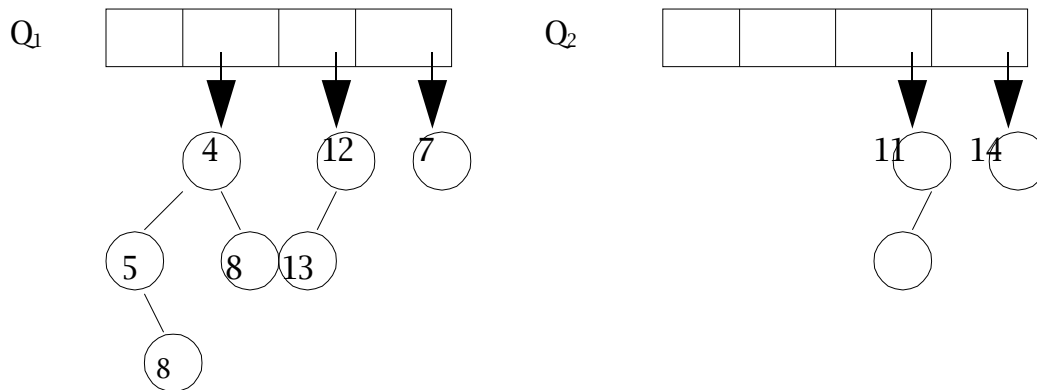
Note that the size m of array a is $m = \lfloor \log_2 n \rfloor + 1$. There is no ordering condition on the keys in the roots of the binomial trees. Obviously the minimum is found in $O(\log n)$ time. The melding of two binomial trees can be done in $O(1)$ time by just comparing the roots and linking them.

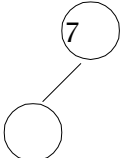
Example



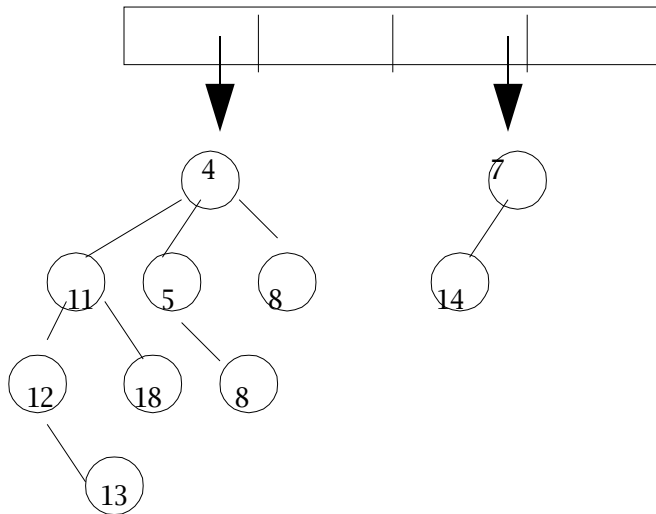
The meld operation for two binomial queues can be implemented like a binary addition. Let the two binomial queues to be melded be Q_1 and Q_2 . We proceed with melding the 0-th binomial trees, first binomial trees, second binomial trees, and so forth, with carries to higher positions.

Example $7 = (0111)_2$ and $3 = (0011)_2$. Then $10 = (1010)_2$.



Since we have a carry  , there are three binomial trees of size 2. An

arbitrary one can be put at position 1 and the other two are melded and carried over to the next position.. Now there are two binomial trees of size 4, which are melded to produce a carry to the next position. Finally we have the following.



The time for meld is $O(\log n)$.

The operation “delete_min” is described as follows: Perform find_min first. Deleting it will cause the tree containing it to be dispersed into fragments, resulting in another binomial queue. Now meld the original binomial queue from which the tree containing the minimum was removed with the new binomial queue. Time is $O(\log n)$.

The operations decrease_key and increase_key can be implemented in $O(\log n)$ time, since we can repeatedly swap the element updated with ancestors or descendants, and the longest path length is $O(\log n)$.

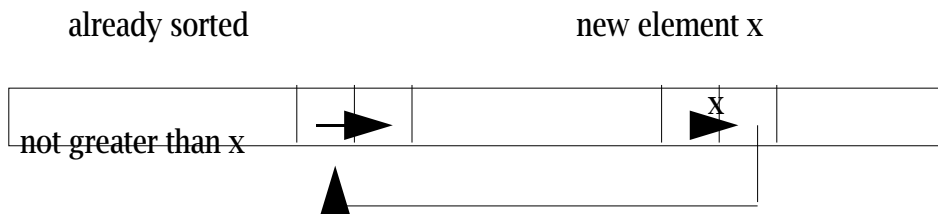
The Fibonacci heap is a variation of the binomial queue, invented by Fredman and Tarjan in 1984. The remarkable aspect of this data structure is that we can perform n delete_min and m decrease_key operations in $O(m + n \log n)$ time.

2. Sorting

Sorting is to sort items with their keys in increasing order. For simplicity we only consider keys to be rearranged. Computing time will be measured mainly by the number of comparisons between keys.

2.1 Insertion sort

This method works on an one-dimensional array (or simply array). A new element is inserted into the already sorted list by shifting the elements to the right until we find a suitable position for the new element. See below.



This method takes about $n^2/4$ comparisons and not very efficient except for small n .

2.2 Selection sort

This method repeatedly selects the minimum from the remaining list. If we implement this method only using an array, we need $n-1$ comparisons for the minimum, $n-2$ comparisons for the second minimum, and so forth, taking about $n^2/2$ comparisons, not efficient.

We use a priority queue to select successive minima. The abstract form of algorithm is described as follows:

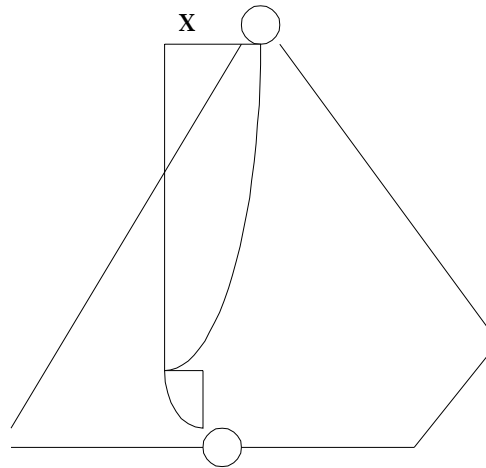
1. Construct a priority queue

2. Initialise the list for the sorted result
3. Repeatedly select and delete the minimum from the queue and append it to the list

The most efficient selection sort is heapsort invented by Williams and Floyd in 1964.

1. build_heap;
2. **for** i:=n downto 2 **do begin**
3. swap(a[1], a[i]);
4. siftup(1,i-1)
5. **end.**

Since we need two comparisons to come down the heap after the swap operation, the number of comparisons for one siftup is bounded by $2\log(n)$. Hence the computing time is $O(n\log n)$. For random keys, the swapped element x put at the root is likely to come down near the bottom of the tree. Regard the root as a hole. A small gain can be made here by sending the smaller child up along the path towards the bottom, and the finding a suitable position for x by going up from the bottom. We spend about $\log n$ comparisons to come to the bottom, and a few more to go up. See below.



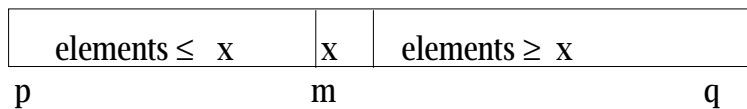
2.3 Exchange sort

Repeatedly exchange two elements until sorted. The most efficient and famous exchange sort is quicksort invented by Hoare in 1961, described below.

1. **procedure** quicksort(p, q);
2. **begin** m : local

3. **if** p<q **then begin**
4. x:=a[p];
5. partition(p, q, m); /* this procedure should be expanded here from next page */
6. quicksort(p, m-1);
7. quicksort(m+1, q);
8. **end**
9. **end;**
10. **begin** {main program}
11. quicksort(1, n)
12. **end.**

partition(p, q, m) is to put all elements not greater than x in a[p .. m-1], those not smaller than x in a[m+1 .. q] and x at a[m]. See below.



The following procedure for partition is more efficient than the original version made by Hoare, invented by Takaoka in 1984.

1. **procedure** partition(p, q, m);
2. **begin**
3. i:=p; j:=q+1;
4. **while** i < j **do begin**
5. **repeat**
6. j:= j-1;
7. **if** i = j **then goto** 10
8. **until** a[j] < x; /* this < can be \leq */
9. a[i]:= a[j];
10. **repeat**
11. i:= i+1;
12. **if** i = j **then goto** 10
13. **until** a[i] > x; /* this > can be \geq */
14. a[j]:= a[i]
15. **end;**
16. 10: a[i] := x; m:= i
17. **end;**

The parameter m should be of var type. The behaviour of partition can be depicted by the following.

1	2	3	4	5	6	7	8	9	10
41	24	76	11	45	64	21	69	19	36
i		i							j j

1	2	3	4	5	6	7	8	9	10
36	24	19	11	21	41	64	69	45	76
m									

j stops at 10 and 36 is put at 1. Next i stops at 3 and 76 is put at 10, etc. This procedure is particularly efficient when there are many identical elements. When we find $x = a[i]$ or $x = a[j]$, we just keep going within the inner loop, whereas swapping takes place in this situation in the original partition by Hoare.

Now let $T(n)$ be the number of comparisons between the input data, not including those between control variables. Then we have the following recurrence.

$$T(1) = 0$$

$$T(n) = n - 1 + (1/n) \sum_{i=1}^{n-1} (T(i) + T(n-i)).$$

We assume that x falls on the range $1 .. n$ with equal probability $1/n$. Partition above takes $n-1$ comparisons whereas Hoare's takes $n+1$ comparisons in the worst case. Similarly to the analysis for binary search trees, we have $T(n)$ is nearly equal to $1.39n \log_2 n$.

Ironically the worst case of quicksort occurs when the given list is already sorted, taking $O(n^2)$ time. This happens because we choose the value of x at line 4 in procedure quicksort, which is called the pivot, to be the leftmost element. To prevent the above mentioned worst case, we can take the median of $a[p]$, $a[(p+q) \div 2]$ and $a[q]$. It is known that if the pivot comes near to the center, the performance is better. The above choice also contribute in this aspect. There are many more strategies on how to choose good pivots.

2.4 Merge sort

Mergesort sorts the list by repeatedly merging the sublists from the single elements at the beginning until there is only one sorted list.

Example

41	24	76	11	45	64	21	69	stage 0
24	41	11	76	45	64	21	69	stage 1
11	24	41	76	21	45	64	69	stage 2
11	21	24	41	45	64	69	76	stage 3

The procedure mergesort is given below.

1. **procedure** mergesort(p, q);
2. **begin** m : local
3. **if** p < q **then begin**
4. m := (p+q) div 2;
5. mergesort(p, m);
6. mergesort(m+1, q);
7. merge(p, m+1, q+1)
8. **end**
9. **end;**
9. **begin** { main program }
10. mergesort(1. n)
11. **end.**

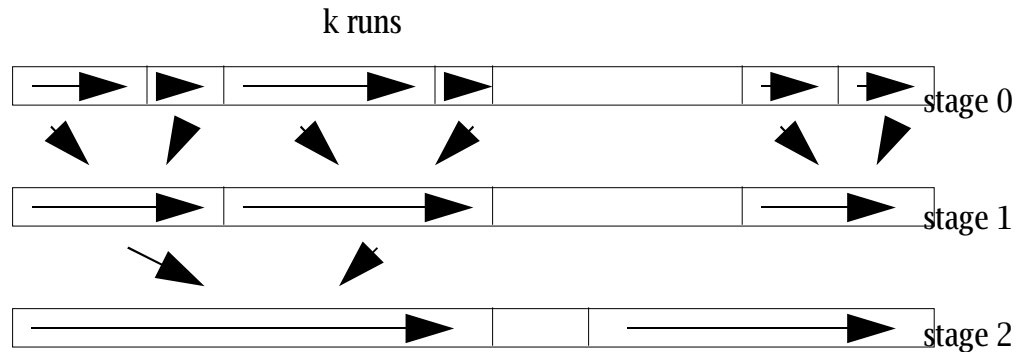
procedure merge(p, m, q) is to merge sorted arrays a[p .. r-1] and a[r .. q-1] described below.

1. **procedure** merge(p, r, q);
2. **begin**
3. i:=p; j:=r; k:=p;
4. **while** i < r **and** j < q **do begin**
5. **if** a[i] <= a[j] **then begin** b[k] := a[i]; i:=i+1 **end**
6. **else begin** b[k] := a[j]; j:=j+1 **end;**
7. k:=k+1
8. **end;**
9. **while** i < r **do begin**

Solving this recurrence yields $T(n) = n \log_2 n - n + 1$. In terms of comparisons, mergesort is one of the best. The drawback is that we need an extra array b , and also the number of assignment statements executed is greater than that of other methods.

Other merits of mergesort is that it is suitable for external sort which uses external merge sort, briefly described below.

Natural mergesort scans the list from left to right to find ascending runs spending $O(n)$ time, and then repeatedly merges two adjacent runs. See below.



After $\log_2 k$ stages, we are done. At each stage we consume at most $n-1$ comparisons. Thus the computing time $O(n \log k)$, precisely speaking $O(n \log(k+1))$ to prevent $T(n)=O(0)$ when $k=1$, in which case $T(n)=O(n)$.

Minimal mergesort, invented by Takaoka in 1996, repeatedly merges two shortest remaining runs after the prescanning. Let $p_i = n_i / n$ where n_i is the length of the i -th run. Let entropy $H(n)$ be defined by

$$H(n) = - \sum_{i=1}^k p_i \log p_i$$

Then minimal mergesort sorts the given list in $O(nH(n))$ time, or precisely speaking $O(n(H(n)+1))$. Note that $0 < H(n) < \log k$.

Example $n_1 = n_2 = 1$. $n_i = 2n_{i-1}$ for $i > 2$. The natural mergesort takes $O(n \log \log n)$ time, whereas minimal mergesort takes $O(n)$ time.

2.5 Radix sort

Radix sort is totally different from other methods. It does not compare the keys, but inspects the digits of the given numbers, hence the name. There are two ways of

inspecting the digits. One is to start from the most significant digit (MSD) and the other from the least significant digit (LSD).

Let us proceed with examples.

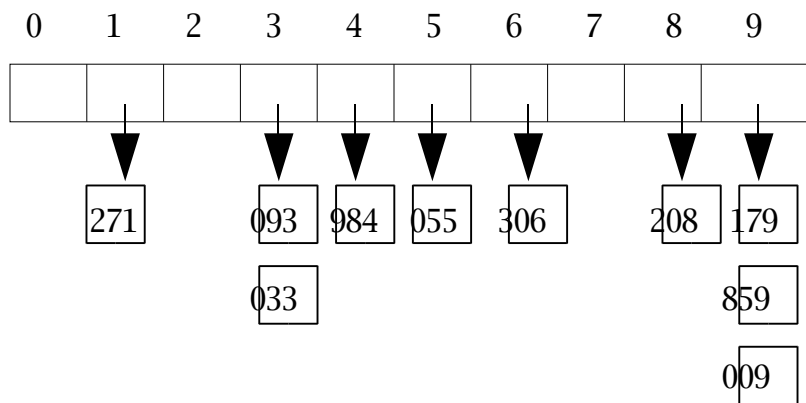
(179, 208, 306, 093, 859, 984, 055, 009, 271, 033)

The MSD method classifies these ten numbers, starting at the first digit into

(093, 055, 009, 033), (179), (208, 271), (306), (859), (984),

in order of 0 .. 9. Empty lists are omitted. We apply the same procedure to each non-empty list for the second digit, third, and so forth. This method is conceptually simple, but not efficient for actual implementation.

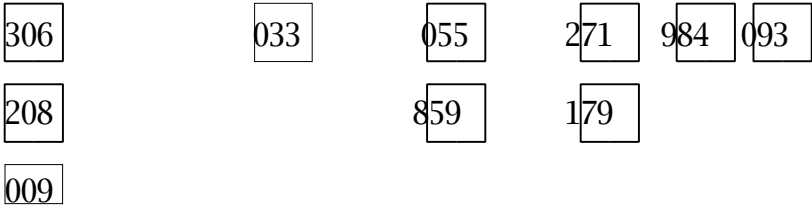
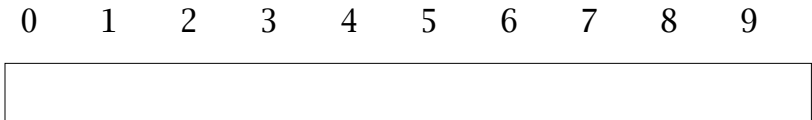
Next we describe the LSD method. We start at the least significant digit. As we scan the list from left to right, we put the numbers in linearly ordered list each linked from the i -th array element if the least significant digit is i . The linearly ordered lists are sometimes called buckets.



The we traverse the lists from left to right and in each list from up to down, and we concatenate them into a single list as follows:

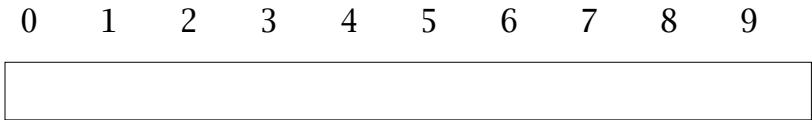
(271, 093, 033, 984, 055, 306, 208, 179, 859, 009)

We do the same using the second least significant digit as follows:



(306, 208, 009, 039, 055, 859, 271, 179, 984, 093)

Finally we have



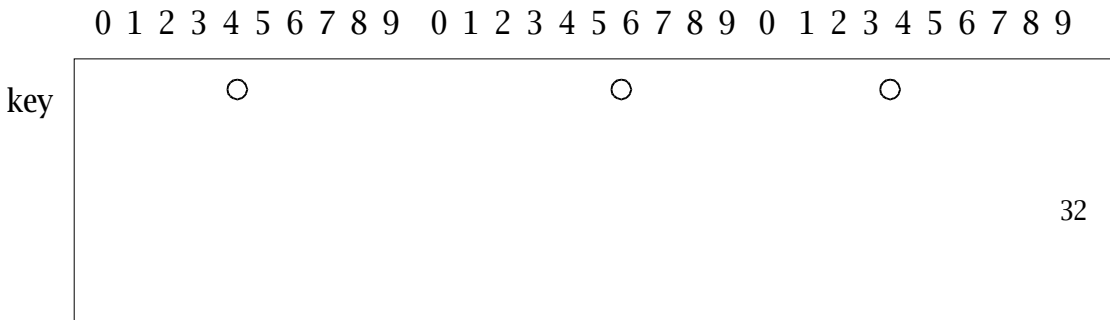
(009, 033, 055, 093, 179, 208, 271, 306, 859, 984)

At the end of i -th stage, numbers are sorted for the last i digits. Computing time is $O(dn)$ where d is the number of digits. The method can be generalized to radix- r number. That is, each number x is contained in an array and expressed as a radix- r number as follows:

$$x = x[d-1] r^{d-1} + \dots + x[1] r + x[0].$$

The computing time then becomes $O(d(n+r))$. In the previous example, $d=3$ and $r=10$. For fixed d and r , this is $O(n)$. When n is large as compared to d and r , this method is very efficient, but when n is small, there will be many empty buckets and time will be wasted checking empty buckets..

This method is also suitable for manual sorting with cards as described below.

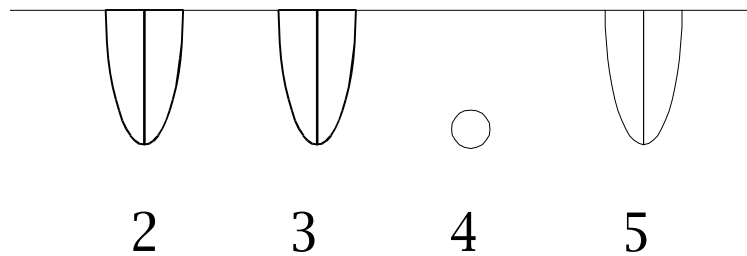


field

4 5 3

data
field

In the above, the key 453 is punched, shown by little circles. The other positions are punched out as shown below. Imagine we have a few hundred cards. By a stick we can pick up cards with keys whose last digits are 3 when we insert it at digit 3 in the last subfield. We do this for 0 .. 9 in the last subfield and put the cards in this order.. Next we do the same for the subfield 2, and so on. The IBM Company made this machine in the ancient times.

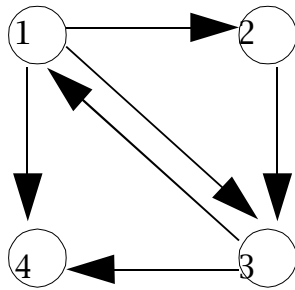


3. Basic Graph Algorithms

3.1 Introduction

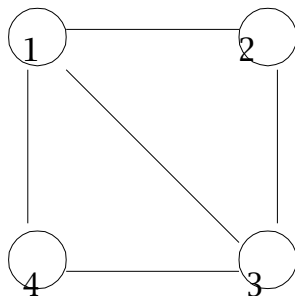
A graph G is defined by $G = (V, E)$ where V is the set of vertices and E is the set of edges. The set E is a subset of the direct product $V \times V$. Conventionally we express as $|V| = n$ and $|E| = m$. For convenience of computer implementation, we express vertices by integers, that is, $V = \{1, \dots, n\}$. This definition of graph is mathematically complete, but not very good for human consumption. If we express vertices by points and edges by arrows, we can visualise the given graph.

Example 1. $G = (V, E)$, $V = \{1, 2, 3, 4\}$, $E = \{(1, 2), (1, 3), (1, 4), (2, 3), (3, 1), (3, 4)\}$.



In the above definition we distinguish between (u, v) and (v, u) , that is, we define directed graphs. If we ignore direction over edges, we have an undirected graph, in which we regard (u, v) and (v, u) as the same thing, denoted by $[u, v]$.

Example 2. If we ignore direction in Example 1, we have $G = (V, E)$, where $V = \{1, 2, 3, 4\}$ and $E = \{[1, 2], [1, 3], [1, 4], [2, 3], [3, 4]\}$.



A path is a sequence of edges $(v_0, v_1)(v_1, v_2) \dots (v_{k-1}, v_k)$, which is called a path from v to v of length k . For convenience we define the null for the case of $k = 0$. There is the null path from v to v for any vertex v . For simplicity the above path is expressed as (v_0, v_1, \dots, v_k) . If all v_i 's are distinct in the path, it is called a simple path. If $v_0 = v_k$, for $k > 0$, this path is called a cycle. If all vertices are distinct except for the first and the last in a cycle, this cycle is called a simple cycle. In an undirected graph similar concepts are defined using $[u, v]$ in stead of (u, v) .

Example 3. In Example 1, $(1, 2, 3, 4)$ is a simple path, $(1, 2, 3, 1)$ is a simple path, $(1, 2, 3, 1, 4)$ is a non-simple path, and $(1, 2, 3, 1, 3, 1)$ is a non-simple cycle.

Example 4. For graph in Example 2, $(1, 2, 3, 4)$ is a simple path and $(1, 2, 3, 4, 1)$ is a simple cycle.

Now we define reachability. If there is a path from u to v we say v is reachable from u and write as $u \rightarrow v$. Since there is a null path from v to v for any v , we have $v \rightarrow v$. If u

$\rightarrow v$ and $v \rightarrow u$, we say u and v are mutually reachable and write as $u \leftrightarrow v$. This relation \rightarrow is an equivalence relation on the set V . The equivalence classes defined by \rightarrow are said to be strongly connected (sc) components of the graph. The vertices in the same equivalence class are mutually reachable. In an undirected graph, there is no distinction between \rightarrow and \rightarrow . Equivalence classes are called connected components.

Example 5 For the graph in Example 1, we have sc-components $\{1, 2, 3\}$ and $\{4\}$. For the graph in Example 2, V itself is a single connected component.

Now we investigate into data structures suitable for graphs to be processed by a computer.

Let

$$\text{OUT}(v) = \{ w \mid (v, w) \text{ is in } E \}$$

$$\text{IN}(v) = \{ u \mid (u, v) \text{ is in } E \}.$$

$|\text{OUT}(v)|$ ($|\text{IN}(v)|$) is called the out (in) degree of v . $\text{OUT}(v)$ (or $\text{IN}(v)$) is called the adjacency list of v . The adjacency list expression of the given graph takes $O(m+n)$ space. We can exhaust edges from each vertex in $O(1)$ time per edge. It takes $O(n)$ time in the worst case to check whether there is an edge from u to v .

The next data structure is an adjacency matrix $A = [a_{ij}]$, where

$$a_{ij} = \begin{cases} 1, & \text{if there is an edge } (i, j) \\ 0, & \text{otherwise.} \end{cases}$$

In this expression, we can check if (i, j) is an edge in $O(1)$ time, although we need $O(n^2)$ space. If the graph is sparse, that is, $m \ll n^2$, this expression by adjacency lists is expensive.

Example 6. For the graphs in Examples 1 and 2, we have

$$A = \begin{matrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{matrix}$$

$$A = \begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{matrix}$$

3.2 Depth-first Search and Breadth-first Search

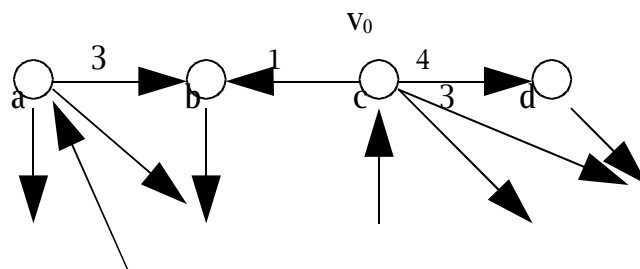
We consider the problem of visiting all vertices in a graph starting from some designated vertex. There are two representative search methods; depth-first (DFS) and breadth-first (BFS). In the following we give the visit number to each vertex.

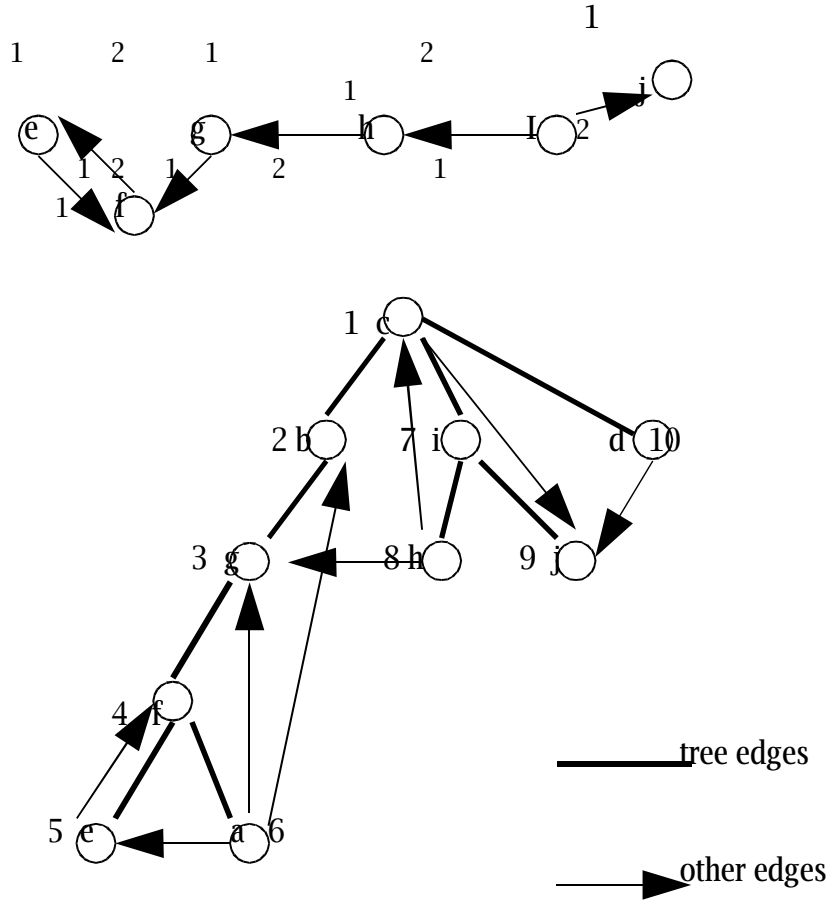
Depth-first search. We start from a vertex and go straight forward to the dead end. Then come back to the previous vertex and again go straight and so forth. We keep track of visited vertices and traversed and examined edges so that we will not examine those things again. If we come back to the starting vertex and have exhausted all edges, we finish the search from the starting vertex. If there are still unvisited vertices, we hop to an unvisited vertex and repeat the same procedure. The algorithm follows.

1. **procedure** DFS(v);
2. **begin** mark v “visited”;
3. $N[v]:=c$; $c:=c+1$;
4. **for** each w in OUT(v) **do**
5. **if** w is “unvisited” **then begin**
6. Add edge (v, w) to T;
7. DFS(w)
8. **end**
9. **end**;
10. **begin** {main program}
11. $T := \emptyset$; {T is the set of edges actually traversed}
12. $c:=1$; {c is the counter for visit numbers}
13. **for** v in V **do** mark v “unvisited”;
14. **while** there is an unvisited v in V **do** DFS(v)
15. **end**.

$N\{v\}$ is the visit number of vertex v. If there is an edge to an unvisited vertex w, we go to w by the recursive call line 7. If we exhaust all edges or OUT(v) is empty from the beginning, we finish DFS(v) and go back to the previous vertex. If we finish one DFS at line 14, the traversed edges which are recorded in T form a tree. If we do DFS by changing v at line 14, we form several trees, that is, a forest, which is called a depth-first spanning forest of the given graph G.

Example 1. Apply the algorithm to the following graph starting from v_0 . The numbers on edges show the order in which vertices are placed in OUT(v) for each v. Note that edges are examined in that order. The edges in T are shown by solid arrows. Dotted arrows are non-traversed edges, that is, only examined. The numbers beside the circles show the visit numbers. In this example we have only one tree in the forest.





After depth-first search, edges are classified into four categories.

- (1) Tree edges; edges recorded in T which are actually traversed.
- (2) Forward edges; non-traversed edges which go from ancestors to descendants.
- (3) Back edges; non-traversed edges which go from descendants to ancestors.
- (4) Cross edges; non-traversed edges which go between vertices that are neither ancestors nor descendants of one another.

Note that a cross edge goes from a vertex whose visit number is greater to a vertex whose visit number is smaller.

Example (c, j) is a forward edge. (h, g), (d, j) and (a, e) are cross edges. (e, f), (a, g), (a, b) and (h, c) are back edges.

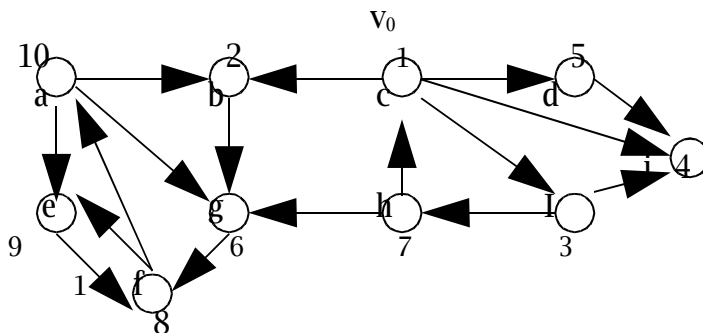
Breadth-first search. In depth-first search, we use a stack implicitly by the recursive call to keep track of the search history. If we use a queue, or a first-in-first-out data structure, we will have the following breadth-first algorithm.

1. procedure BFS(v);

2. begin'
3. $Q := \{v\}$; /* Q is a queue */
4. while $Q \neq \emptyset$ do begin
5. $v \leftarrow Q$; /* take out from Q */
6. Mark v "visited";
7. $N[v] := c$; $c := c + 1$;
8. for each w in $OUT(v)$ do
9. if w is "unvisited" and w is not in Q then
10. $Q \leftarrow w$ /* push into Q */
11. end
12. end;
13. begin {main program}
14. for v in V do mark v "unvisited";
15. $c := 1$;
16. while there is an unvisited v in V do BFS(v)
17. end.

As a by-product we can compute shortest distances to vertices reachable from the starting vertex.. If we record traversed edges, we can compute shortest paths as well.

Example 2. If we apply BFS to the graph in Example 1, we have the following visit numbers. After we process $OUT(b)$, the status of Q is given by $Q = (i, j, d, g)$.



3.3 Strongly Connected Components

The concept of depth-first search is applied to find strongly connected components in this section. Let the sets of strongly connected components be V_i ($i=1, \dots, k$). Let the set of edges E_i be defined by $E_i = \{ (u, v) \mid u \text{ and } v \text{ are in } V_i \}$. Then we have the following lemma.

Lemma. Graph $(V_i, E_i \cap T)$ form a tree.

Example. In Example 1, we have $V = \{b, g, f, e, a\}$, $V = \{j\}$, $V = \{d\}$, $V = \{c, i, h\}$. Observe that each sc-component forms a tree in the depth-first spanning forest.

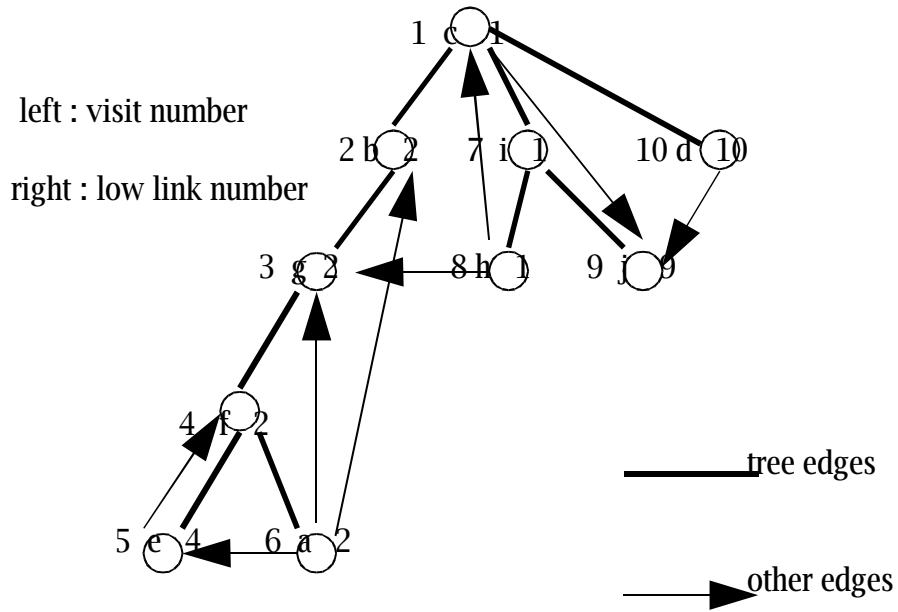
If we can know the root of each sc-component, we can harvest it when we finish DFS from that vertex. If we can reach from v to vertex with a lower visit number, v can not be the root of an sc-component. We record this reachability in array L . Also accumulated candidates for sc-components are recorded in $STACK$, a stack data structure, that is, first-in-last-out data structure. The algorithm follows.

1. **procedure** CONNECT(v);
2. **begin** Mark v “visited”;
3. $N[v]:=c$; $c:=c+1$;
4. $L[v]:=N[v]$; $STACK \leftarrow \{v\}$; /* push v to $STACK$ */
5. **for** each w in $OUT(v)$ **do**
6. **if** w is “unvisited” **then begin**
7. CONNECT(w);
8. $L[v] := \min\{ L[v], L[w] \}$;
9. **end**
10. **else if** $N[w] < N[v]$ **and** w is in $STACK$ **then**
11. $L[v] := \min\{ N[w], L[v] \}$;
12. **if** $L[v] = N[v]$ **then begin**
13. Take vertices out of $STACK$ until v and write them
14. **end**;
15. **begin** {main program}
16. $c:=1$; $STACK := \emptyset$;
17. **for** each v in V **do** mark v “unvisited”;
20. **while** there is an “unvisited” vertex v **do** CONNECT(v)
21. **end**.

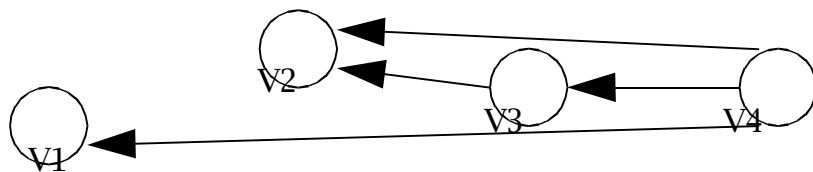
Example. If we apply CONNECT to the graph in Example 1, we have

$$V_1 = \{a, e, f, g, b\}, V_2 = \{j\}, V_3 = \{d\}, V_4 = \{h, i, c\}$$

in this order. In the following figure, visit numbers are attached to the right of circles and those of L are attached to the right. L is called the low link.



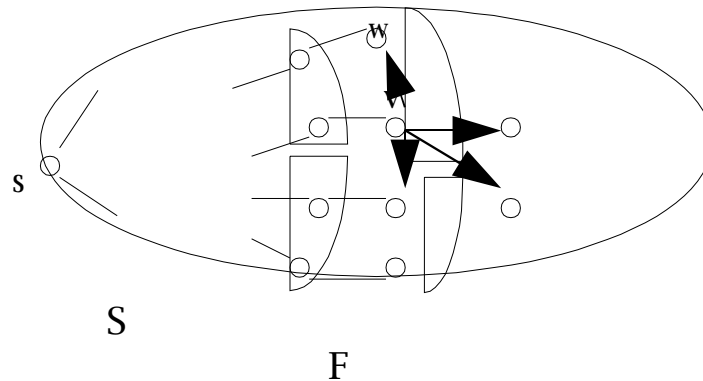
If we crush each sc-component into a single vertex and edges connecting them into single edges, we have an acyclic graph as shown below.



4 Shortest Path Algorithms

4.1 Single Source Problem -- Dijkstra's Algorithm

We consider the problem of computing the shortest paths from a designate vertex s , called the source, to all other vertices in the given graph $G = (V, E)$. For simplicity we assume all other vertices are reachable from s . We maintain three kinds of sets, S , F , and $V-S-F$, called the solution set, the frontier set, and the unknown world. See below.



The set S is the set of vertices to which the algorithm computed the shortest distances. The set F is the set of vertices which are directly connected from vertices in S . The set $V-S-F$ is the set of all other vertices. We maintain the distance to each vertex v , $d[v]$, which is the shortest distance to v if v is in S . The distance $d[v]$ for v in F is defined as follows:

$d[v]$ = distance of the shortest path that lies in S except for the end point v .

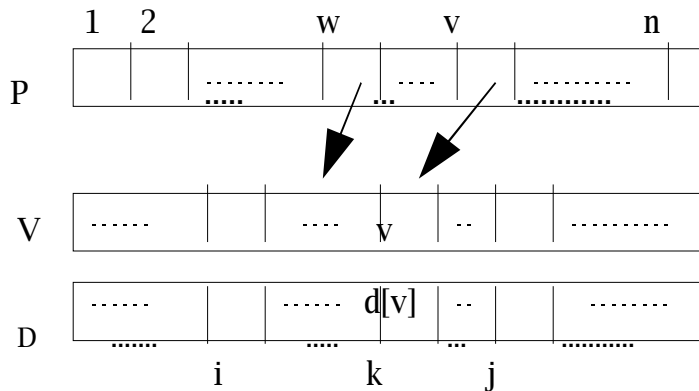
If we find the minimum such $d[v]$ for v in F , there is no other route to v which is shorter. Thus we can include v in S . After this we have to update the distances to other vertices in F , and set up distances to vertices in $V-S-F$ that are directly connected from v and include them in F . Let $OUT(v) = \{ w \mid (v, w) \text{ is in } E \}$.

1. $S := \emptyset$;
2. Put s into S ; $d[s] := 0$;
3. **for** v in $OUT(s)$ **do** $d[v] := L(s, v)$; $\{ L(u, v) \text{ is the length of edge } (u, v) \}$
4. $F := \{ v \mid (s, v) \text{ is in } E \}$;
5. **while** F is not empty **do begin**
6. $v := u$ such that $d[u]$ is minimum among u in F
7. $F := F - \{v\}$; $S := S \cup \{v\}$;

8. **for** w in $OUT(v)$ **do**
9. **if** w is not in S **then**
10. **if** w is in F **then** $d[w] := \min \{ d[w], d[v]+L(v, w) \}$
11. **else begin** $d[w] := d[v]+L(v, w)$; $F := F \cup \{w\}$ **end**
12. **end.**

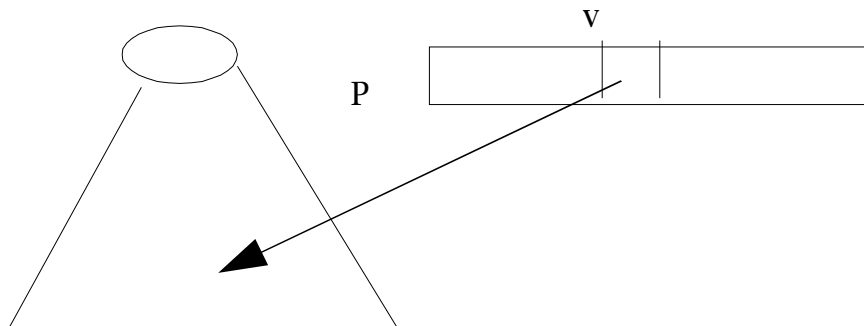
Let the set V of vertices be given by the set of integer $\{1, \dots, n\}$. The simplest way to implement the data structures is that we maintain the distances in array d and the set S in Boolean array S ; $S[v] = \text{true}$ if v is in S , $S[v] = \text{false}$ otherwise. The distances to v in $V-S-F$ is set as $d[v] = \infty$. The set F can be identified by element v such that $S[v] = \text{false}$ and $d[v] = \infty$. Line 6 can be implemented by scanning array d and testing the Boolean condition S , taking $O(n)$ time. Lines 8-11 can be implemented by scanning the list $OUT(v)$, taking $O(n)$ time in the worst case. The overall time will be $O(n^2)$.

A slightly better implementation is described next. The solution set S is maintained in array V with indices $1 \dots I$. The set F is with indices $I+1 \dots j$. The contents of V are actual vertices. The array P is for pointing to the positions of vertices in V . See below.



We find the minimum $d[v]$ at line 6, which is found at k , and swap $(V[i], D[i])$ with $(V[k], D[k])$ and increase I by 1. The corresponding element of P is updated. We also add new elements to V and D for inclusion in F and increase j accordingly. The pointers to the new elements in F will be set up in P . This implementation is efficient when the size of F is not large as in planar graphs such as maps. Also this implementation is a good introduction to the more sophisticated implementation with priority queues.

An implementation with a binary heap is described next. The idea is similar to the above. The pairs $(v, d[v])$ are maintained in a binary heap with $d[v]$ as a key. See below.

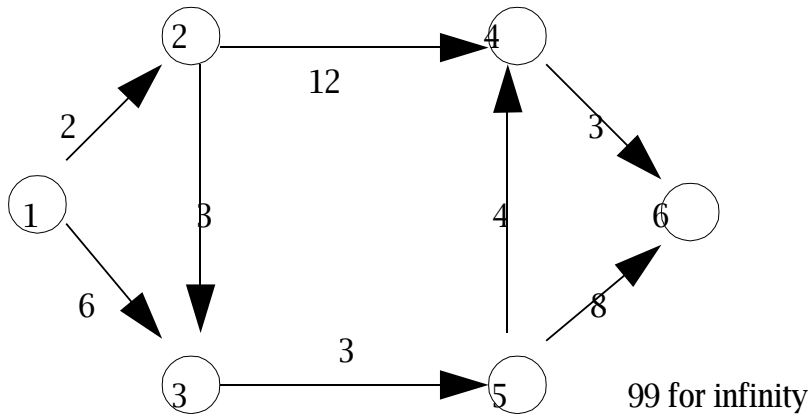


$v, d[v]$

The n minimum operations at line 6 will take $O(n \log n)$ time. Suppose there are m edges. Then the total decrease_key operations and insert operations at lines 7-11 will take $O(m \log n)$ time. Thus the total running time is $O(m \log n)$, since if we can reach all vertices from s , we can assume $n-1 \leq m$. This implementation is efficient when the graph is sparse such as planar graphs where $m = O(n)$. The time becomes $O(n \log n)$ in this case.

If we use the heap as a priority queue, we can solve the problem in $O(m+n \log n)$ time.

Example Let us trace the changes of array d for the following graph.



0	2	6	99	99	99	$v=2, S=\{1\}, F=\{2, 3\}$
1	2	3	4	5	6	

0	2	5	14	99	99	$v=3, S=\{1, 2\}, F=\{3, 4\}$
---	---	---	----	----	----	-------------------------------

0	2	5	14	8	99	$v=5, S=\{1, 2, 3\}, F=\{4, 5\}$
---	---	---	----	---	----	----------------------------------

--	--	--	--	--	--	--

0 2 5 12 8 16 $v=4, S=\{1, 2, 3, 5\}, F=\{4, 6\}$

0	2	5	12	8	15
---	---	---	----	---	----

 $v=6, S=\{1, 2, 3, 5, 4\}, F=\{6\}$

0	2	5	12	8	15
---	---	---	----	---	----

4.2 Transitive Closure and the All Pairs Shortest Path Problem

Let A be the adjacency matrix of the given graph G . That is

$$A[i, j] = 1, \text{ if } (i, j) \text{ is in } E \\ 0, \text{ otherwise.}$$

The reflexive-transitive closure of A , A^* , is defined as follows:

$$A^*[i, j] = 1, \text{ if } j \text{ is reachable from } i \\ 0, \text{ otherwise.}$$

The transitive closure of A , A^+ , is defined as follows:

$$A^+[i, j] = 1, \text{ if } j \text{ is reachable from } i \text{ over a path of at least one edges} \\ 0, \text{ otherwise.}$$

The computations of both closures are similar. In the following we describe the transitive closure. The algorithm was invented by Warshall in 1962.

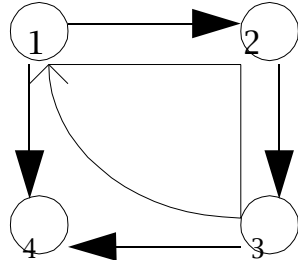
1. **for** $i:=1$ **to** n **do** **for** $j:=1$ **to** n **do** $C[i, j] := A[i, j]$;
2. **for** $i:=1$ **to** n **do** $C[i, i] := C[i, i] + 1$
3. **for** $k:=1$ **to** n **do**
4. **for** $i:=1$ **to** n **do** **for** $j:=1$ **to** n **do**
5. $C[i, j] := C[i, j] \vee (C[i, k] \wedge C[k, j])$

Theorem. At the end of the algorithm, array C gives the transitive closure A^* .

Proof. Theorem follows from mathematical induction on k . We prove that $C[i, j] = 1$ at the end of the k -th iteration if and only if j is reachable from i only going through some of the via set $\{1, 2, \dots, k\}$ except for endpoints i and j . The basis of $k=0$ is clear since the via

set is empty. Now the theorem is true for $k-1$. We observe that we can go from i to j via $\{1, 2, \dots, k\}$ if and only if we can go from i to j via $\{1, \dots, k-1\}$ or we can go from i to k via $\{1, \dots, k-1\}$ and from k to j via $\{1, \dots, k-1\}$. Thus line 5 correctly computes C for k .

Example. We trace array C for each k for the following graph.



	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
$C =$	1 1 0 1	1 1 0 1	1 1 1 1	1 1 1 1	1 1 1 1
	0 1 1 0	0 1 1 0	0 1 1 0	1 1 1 1	1 1 1 1
	1 0 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1
	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1

For hand calculation, it is convenient to enclose the k -th row and column in the $(k-1)$ th array to get the k -th array. To get the (i, j) element in the k -th array, we go horizontally and vertically to the enclosed axes. If we find 1's in both axes, we can set up 1 if the original element was 0. See below.

$k = 0$

1	1	0	1
0	1	1	0
1	0	1	1
0	0	1	1

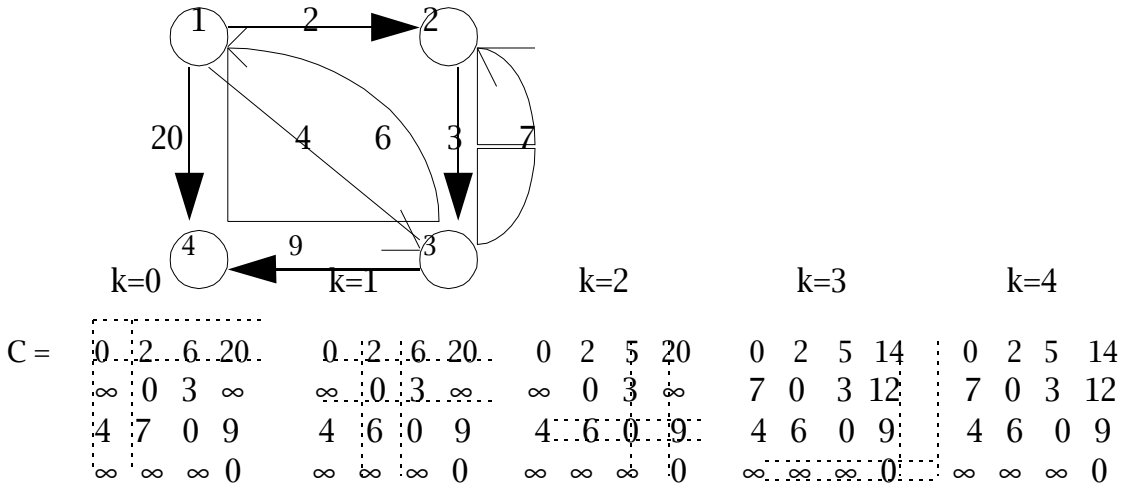
Now we turn to the all pairs shortest path problem. Let $D[i, j]$ be the given distance from vertex i to vertex j . Let $D[i, j]$ be ∞ if there is no edge from i to j . The following algorithm for all pairs problem was invented by Floyd in 1962.

1. for $i:=1$ to n do for $j:=1$ to n do $C[i, j] := D[i, j]$;
2. for $i:=1$ to n do $C[i, i] := 0$;
3. for $k:=1$ to n do
4. for $i:=1$ to n do for $j:=1$ to n do
5. $C[i, j] := \min\{C[i, j], C[i, k] + C[k, j]\}$

Theorem. At the end of the algorithm, array element $C[i, j]$ gives the shortest distance from i to j .

Proof. Similar to that of Theorem 1. We can prove by induction that $C[i, j]$ at the end of the k -th iteration gives the distance of the shortest path from i to j that lies in the via set $\{1, \dots, k\}$ except for endpoints i and j . The details are left with students.

Example. We trace the matrix C at each iteration by k for the following graph.



The axes are enclosed here also to ease calculations. Specifically $C[3, 2]$ at $k=1$ is obtained by $6 = \min\{7, 4+2\}$ in C at $k=0$.

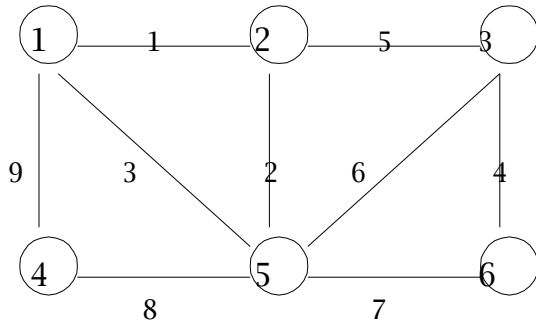
As we see from the above two algorithms, they are very similar. In fact, these two problems can be formulated by a broader perspective of semiring theory. That is, Boolean algebra and the distance calculation both satisfy the axioms of semiring. The general algorithm will calculate the closure of a given matrix over the semiring. See for more details the book “Design and Analysis of Computer Algorithms” by Aho, Hopcroft and Ullman published in 1974.

5 Minimum Cost Spanning Trees

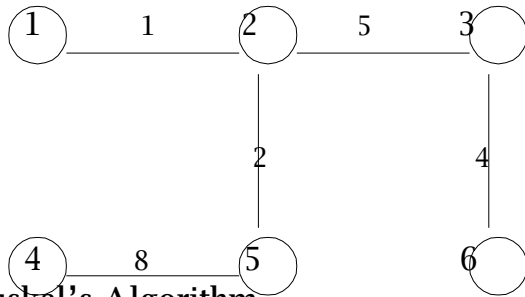
5.1 Introduction

The minimum cost spanning tree (abbreviated as minimum spanning tree) of a given undirected graph G is a spanning tree of G with the minimum cost. A spanning tree is a tree that connect all the vertices. The cost of a spanning tree is the sum of the edge cost. An application of this problem is seen in the city planning that will locate a public facility that satisfy all the needs of the city residents and minimise the cost of the road construction.

Example.

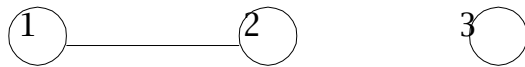


The minimum spanning tree with cost 20 is given by the following.



3.2 Kruskal's Algorithm

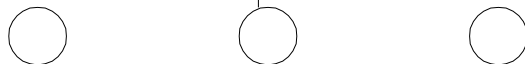
This method belongs to the so-called greedy paradigm. A spanning forest of the graph is a collection of trees that covers all the vertices. Sort the edges in increasing order. It is clear that the shortest edge and the second shortest edge are part of the solution. If the third shortest forms a loop, we skip it. In this way, we gradually grow a spanning forest until we finally have a single spanning tree.

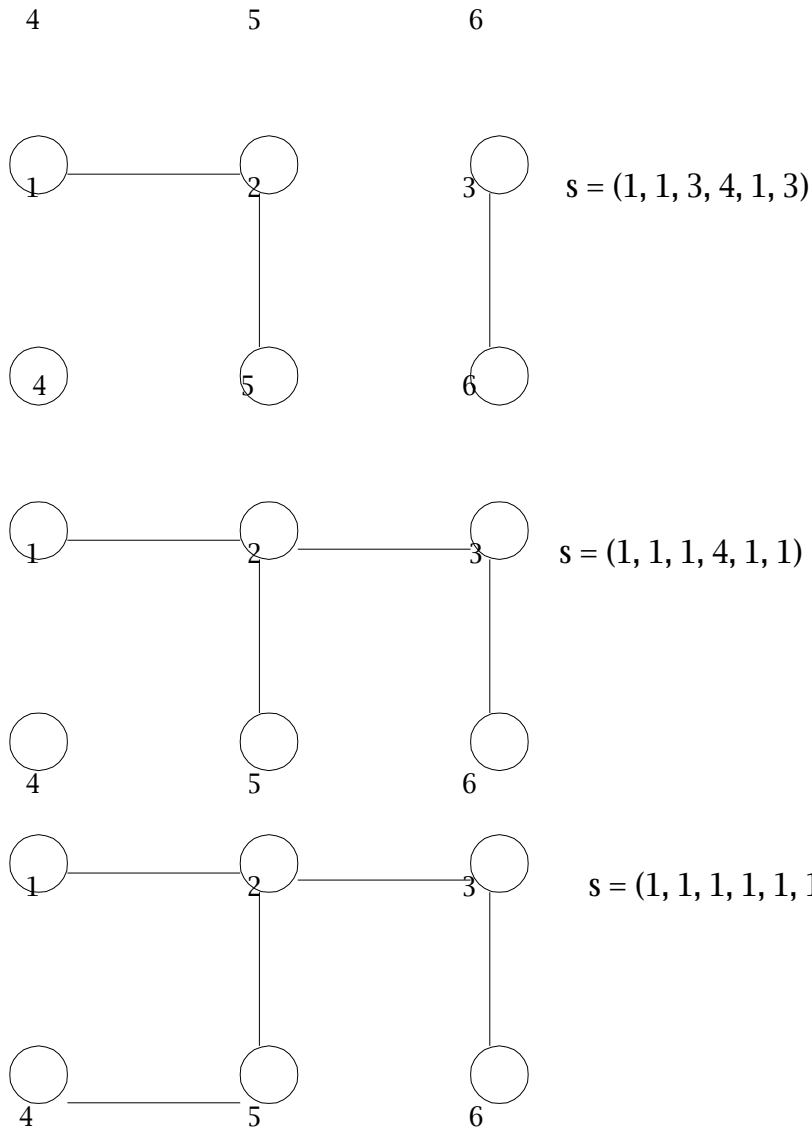


$s = (1, 1, 3, 4, 5, 6)$



$s = (1, 1, 3, 4, 1, 6)$





The spanning forest at each stage is nothing but a collection of disjoint sets of vertices. This can be maintained by a one dimensional array s with the initial setting of $s[i] = i$ for $i = 1, \dots, n$. When we merge two disjoint sets, we adopt the name of the larger set. For example, from the 3rd stage to the 4th stage, we have the following change.

$$s = (1, 1, 3, 4, 1, 3) \Rightarrow s = (1, 1, 1, 4, 1, 1).$$

Now Kruskal's algorithm follows.

1. Sort the set of edges E in increasing order with edge cost as key into list L ;
2. **for** $i:=1$ **to** n **do** $s[i]:=i$;
3. $k:=n$; { k is the number of trees so far obtained}
4. **while** $k>1$ **do begin**

5. $(v, w) \leq L$; { remove the leftmost edge from L and let it be (v, w) }
6. **if** not $(s[v] = s[w])$ **then begin**
7. $T := T \cup \{(v, w)\}$;
8. Change the set name of elements in the smaller to that of the larger;
9. **end**
10. **end.**

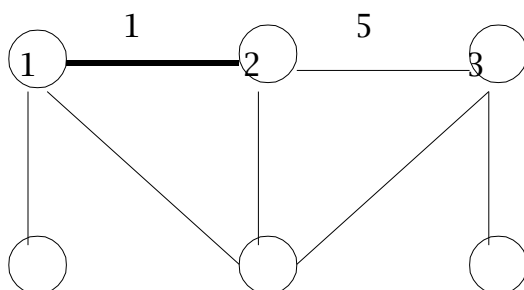
The dominant time complexity is that of sorting, which is $O(m \log m) = O(m \log n)$, since $m \leq n^2$. The overall complexity of merging two sets at line 8 is $O(n \log n)$ by the following observation. Suppose that elements 1, ..., n lives in their rooms separately at the beginning, and move to larger rooms at line 8. Because they move to a larger room each time, the number of each element 's moves is $\log n$. There are n elements. Thus the total cost for moving is $O(n \log n)$.

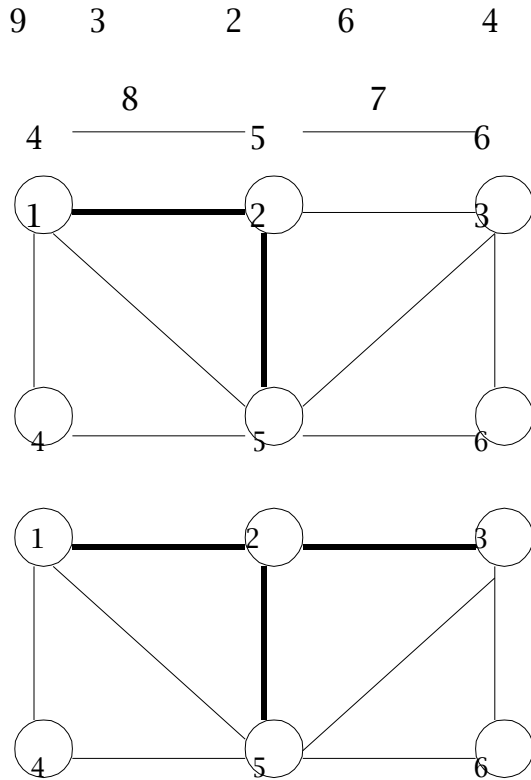
3.3 Prim's Algorithm

This algorithm is similar to Dijkstra's algorithm for shortest paths. In stead of growing a forest in Kruskal's algorithm, we grow a single tree starting from an arbitrary vertex. We maintain the frontier set F of vertices which are adjacent from the tree so far grown. Each member of F has a key which is the cost of the shortest edge to any of vertex in the tree. Each time we expand the tree, we update F and the key values of elements in F. Now Prim's algorithm follows.

1. $S := \emptyset$;
2. Put s into S;
3. **for** v in OUT(s) **do** $c[v] := L(s, v)$;
4. $F := \{v \mid (s, v) \text{ is in } E \}$;
5. **while** $|S| < n$ **do begin**
6. $v := u$ such that $c[u]$ is minimum among u in F;
7. $F := F - \{v\}$; $S := S \cup \{v\}$;
8. **for** w in OUT(v) **do**
9. **if** w is not in S **then**
10. **if** w is in F **then** $c[w] := \min\{ c[w], L(v, w) \}$
11. **else begin** $c[w] := L(v, w)$; $F := F \cup \{w\}$
12. **end.**

Now the initial part of trace for our example follows with bold lines for the tree.





Similarly to Dijkstra's algorithm, if we use a binary heap for F, the time will be $O(m \log n)$ and it will be $O(m + n \log n)$ if we use a Fibonacci heap.

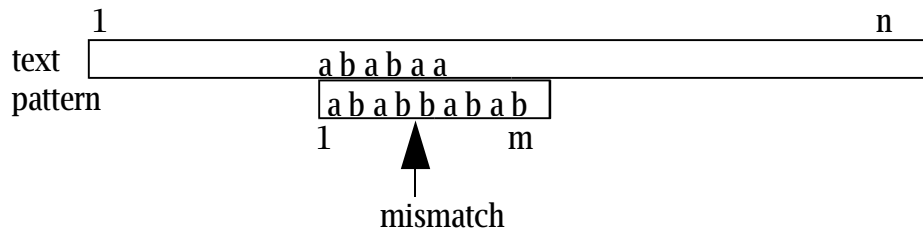
Lecture Notes on Pattern Matching Algorithms

1. Introduction

Pattern matching is to find a pattern, which is relatively small, in a text, which is supposed to be very large. Patterns and texts can be one-dimensional, or two-dimensional. In the case of one-dimensional, examples can be text editor and DNA analysis. In the text editor, we have 26 characters and some special symbols, whereas in the DNA case, we have four characters of A, C, G, and T. In the text editor, a pattern is often a word, whose length is around 10, and the length of the text is a few hundred up to one million. In the DNA analysis, a gene is a few hundred long and the human genome is about 3 billion long.

In the case of two-dimensional, the typical application is a pattern matching in computer vision. A pattern is about $(100, 100)$ and text typically $(1024, 1024)$. Either one-dimensional or two-dimensional, the text is very large, and therefore a fast algorithm to

find the occurrence(s) of pattern in it is needed. We start from a naive algorithm for one-dimensional.



At first the pattern is set to the left end of the text, and matching process starts. After a mismatch is found, pattern is shifted one place right and a new matching process starts, and so on. The pattern and text are in arrays $pat[1..m]$ and $text[1..n]$ respectively.

Algorithm 1. Naive pattern matching algorithm

1. $j:=1;$
2. **while** $j \leq n-m+1$ **do begin**
3. $i:=1;$
4. **while** $(i \leq m)$ **and** $(pat[i]=text[j])$ **do begin**
5. $i:=i+1;$
6. $j:=j+1$
7. **end;**
8. **if** $i \leq m$ **then** $j:=j-i+2$ /* shift the pattern one place right */
9. **else write**("found at ", $j-i+1$)
10. **end.**

The worst case happens when pat and $text$ are all a's but b at the end, such as $pat = aaaaab$ and $text = aaaaaaaaaaaaaaaaaaaaaaaab$. The time is obviously $O(mn)$. On average the situation is not as bad, but in the next section we introduce a much better algorithm. We call the operation $pat[i]=text[j]$ a comparison between characters, and measure the complexity of a given algorithm by the number of character comparisons. The rest of computing time is proportional to this measure.

2. Knuth-Morris-Pratt algorithm (KMP algorithm)

When we shift the pattern to the right after a mismatch is found at i on the pattern and j on the text, we did not utilise the matching history for the portion $pat[1..i]$ and $text[j-i+1..j]$. If we can get information on how much to shift in advance, we can shift the pattern more, as shown in the following example.

Example 1.

1 2 3 4 5 6 7 8 9 10 11 12 13 14

```

text   a b a b a a b b a b a b b a
pattern a b a b b
        a b a b b
          a b a b b
            a b a b b

```

After mismatch at 5, there is no point in shifting the pattern one place. On the other hand we know “ab” repeats itself in the pattern. We also need the condition $pat[3] \neq pat[5]$ to ensure that we do not waste a match at position 5. Thus after shifting two places, we resume matching at position 5. Now we have a mismatch at position 6. This time shifting two places does not work, since “ab” repeats itself and we know that shifting two places will invite a mismatch. The condition $pat[1] \neq pat[4]$ is satisfied. Thus we shift pat three places and resume matching at position 6, and find a mismatch at position 8. For a similar reason to the previous case, we shift pat three places, and finally we find the pattern at position 9 of the text. We spent 15 comparisons between characters. If we followed Algorithm 1, we would spend 23 comparisons. Confirm this.

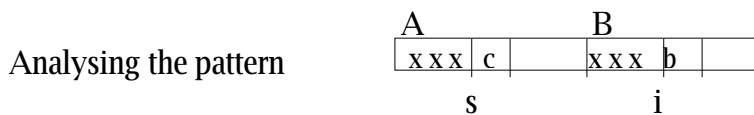
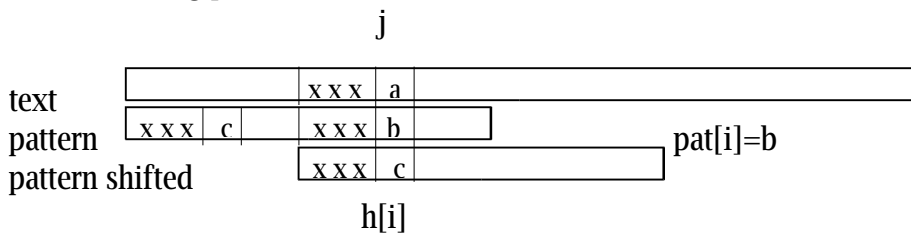
The information of how much to shift is given by array $h[1..m]$, which is defined by

$$h[1] = 0$$

$$h[i] = \max \{ s \mid (s=0) \text{ or } (pat[1 .. s-1] = pat[i-s+1 .. i-1] \text{ and } pat[s] \neq pat[i]) \}$$

The situation is illustrated in the following figure.

Main matching process



The meaning of $h[i]$ is to maximise the portion A and B in the above figure, and require $b \neq c$. The value of $h[i]$ is such maximum s . Then in the main matching process, we can resume matching after we shift the pattern after a mismatch at i on the pattern to position $h[i]$ on the pattern, and we can keep going with the pointer j on the text. In other words, we need not to backtrack on the text. The maximisation of such s , (or minimisation of shift), is necessary in order not to overlook an occurrence of the pattern in the text. The main matching algorithm follows.

Algorithm 2. Matching algorithm

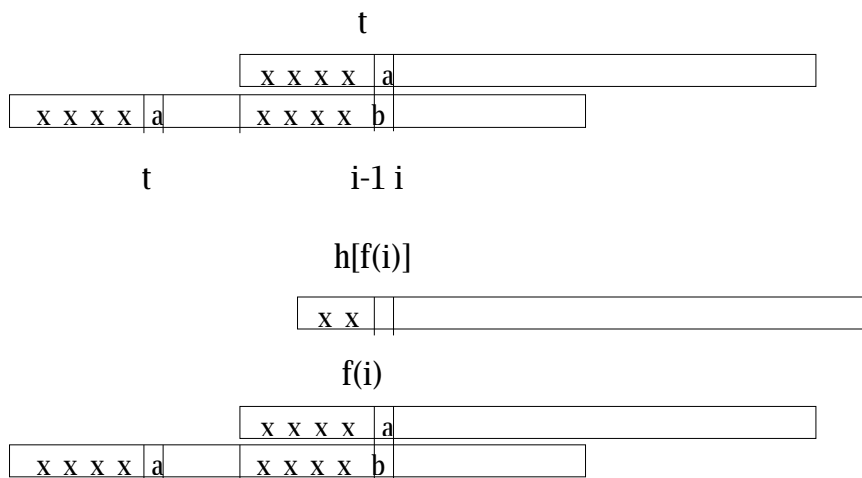
1. $i:=1; j:=1;$
2. **while** ($i \leq m$) **and** ($j \leq n$) **do begin**
3. **while** ($i > 0$) **and** ($pat[i] \neq text[j]$) **do** $i:=h[i];$
4. $i:=i+1; j:=j+1$
5. **end**
6. **if** $i \leq m$ **then write**(“not found”)
7. **else write**(“found at”, $j-i+1$)

Let the function $f(i)$ be defined by

$$f(1) = 0$$

$$f(i) = \max\{ s \mid (1 \leq s < i) \text{ and } (pat[1 .. s-1] = pat[i-s+1 .. i-1]) \}$$

The definitions of $h[i]$ and $f(i)$ are similar. In the latter we do not care about $pat[s] \neq pat[i]$. The computation of h is like pattern matching of pat on itself.



Algorithm 3. Computation of h

1. $t:=0; h[1]:=0;$
2. **for** $i:=2$ **to** m **do begin**
3. /* $t = f(i-1)$ */
4. **while** ($t > 0$) **and** ($pat[i-1] \neq pat[t]$) **do** $t:=h[t];$
5. $t:=t+1;$
6. /* $t=f(i)$ */

7. **if** pat[i]<>pat[t] **then** h[i]:=t **else** h[i]:=h[t]
8. **end**

The computation of h proceeds from left to right. Suppose $t=f(i-1)$. If $pat[t]=pat[i-1]$, we can extend the matched portion by one at line 5. If $pat[t]<>pat[i-1]$, by repeating $t:=h[t]$, we effectively slide the pattern over itself until we find $pat[t]=pat[i-1]$, and we can extend the matched portion. If $pat[i]<>pat[t]$ at line 7, the position t satisfies the condition for h, and so h[i] can be set to t. If $pat[i]=pat[t]$, by going one step by h[t], the position will satisfy the condition for h, and thus h[i] can be set to h[t].

Example. pat = a b a b b

i = 2, at line 7 t=1, pat[1]<>pat[2], f(2) = 1, h[2]:=1

```

      i
    a b a b b
      a b a b b
        t

```

i = 3, t=1 at line 4. pat[1]<>pat[2], t:=h[1]=0, at line 7 t=1, f(3)=1, pat[1]=pat[3], h[3]:=0

```

    a b a b b
      a b a b b

```

i = 4, t=1 at line 4. pat[3]=pat[1], t:=t+1, t=2. At line 7, pat[4]=pat[2], h[4]:=h[2]=1

i = 5, t=2, f(4)=2. At line 4, pat[4]=pat[2], t:=t+1=3. f(5)=3. At line 7, pat[5]<>pat[3], h[5]:=t=3

Finally we have

```

  i | 1  2  3  4  5
-----
pat | a  b  a  b  b
  f | 0  1  1  2  3
  h | 0  1  0  1  3

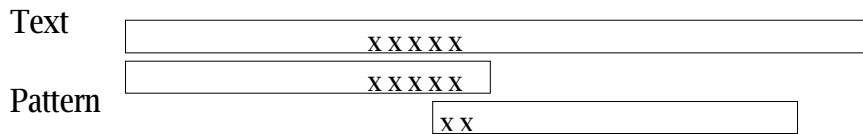
```

The time of Algorithm 2 is clearly $O(n)$, since each character in the text is examined at most twice, which gives the upper bound on the number of comparisons. Also, whenever a mismatch is found, the pattern is shifted to the right. The shifting can not take place more than $n-m-1$ times. The analysis of Algorithm 3 is a little more tricky. Trace the changes on the value of t in the algorithm. We have a doubly nested loop, one by the outer for and the other by while. The value of t can be increased by one at line 5, and $m-1$ times in total, which we regard as income. If we get into the while loop, the value of t is decreased, which we regard as expenditure. Since the total income is $m-1$, and t can not go to negative, the total number of executions of $t:=h[t]$ can not exceed $m-1$. Thus the total time is $O(m)$.

Summarising these analyses, the total time for the KMP algorithm, which includes the pre-processing of the pattern, is $O(m+n)$, which is linear.

A Brief Sketch of the Boyer-Moore Algorithm

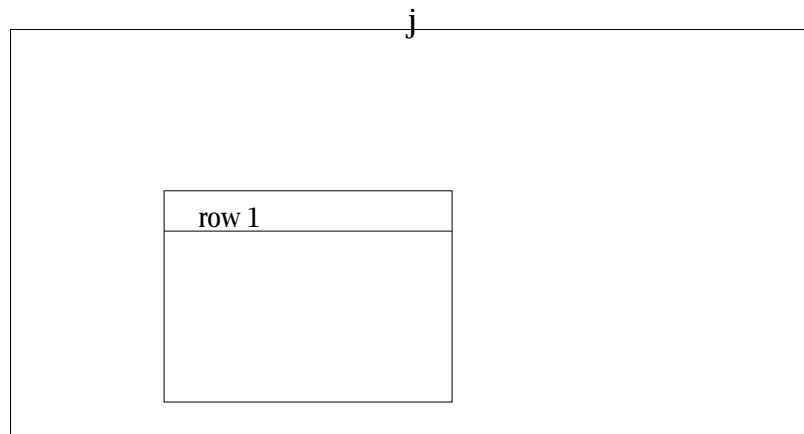
In the KMP algorithm, we started matching process from the left end of the pattern. In the Boyer-Moore algorithm, we start matching from the right end of algorithm. By doing pre-processing on the pattern, we know how much to shift the pattern over the text when a mismatch is detected. See the following figure.

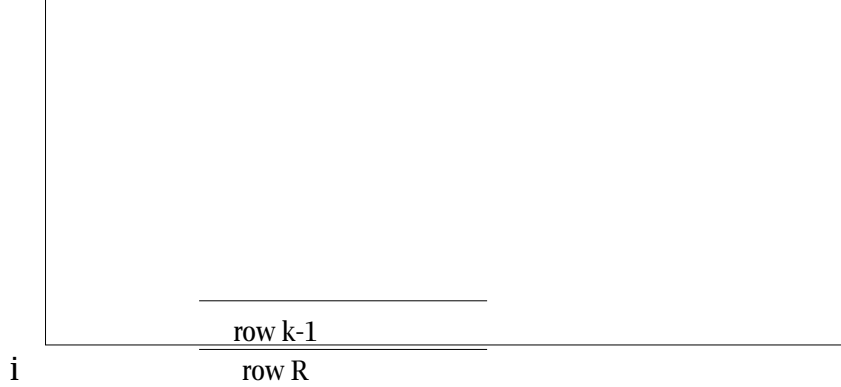


If the matched portion or a part of it exists within the pattern, we make an alignment as shown in the figure, and start the matching process from the right end of the pattern. For a random text and pattern, the probability that the matched portion appears in the pattern is very small, which means that the pattern can move the distance of m , which in turn gives us the time of $O(n/m)$. As the parameter m is in the denominator, this complexity is called sub-linear, much faster than the KMP algorithm for a random text and pattern.

A Brief Sketch of Bird's Algorithm for Two-Dimensional Matching

The Aho-Corasick (AC) algorithm find an occurrence of a pattern of several patterns in $O(n)$ time where n is the length of the text, with pre-processing of the patterns in linear time, that is, in $O(m_1 + \dots + m_l)$ time where there are l patterns. This is called the multiple pattern matching and is a generalization of the KMP method. In Bird's algorithm, we use the AC algorithm horizontally and the KMP algorithm vertically. See the following figure.





We call the m rows of the pattern row 1, ..., row m . Some of the rows may be equal. Thus we can compute the h function of row 1, ..., row m for the vertical pattern matching. We prepare a one-dimensional array $a[1..n]$ to keep track of the partial matching. If $a[j]=k$, it means we have found row 1, ..., row $k-1$ of the pattern in the region of the text from the point (i,j) towards up and left. If the newly found row R is not found to be any row of the pattern, $a[j]$ is set to 1. If row R belongs to the pattern, we use the h function of the vertical pattern. While $R \neq \text{row } k$, we perform $k:=h[k]$.

After this, if R matches row k , $a[j]$ is set to $k+1$. If $a[j]$ becomes $m+1$, the pattern is found. By a similar reasoning to the one-dimensional case, we can show that the time is $O(m^2 + n^2)$, which is linear. This is much better than the exhaustive method with $O(m^2n^2)$, as the data size is enormous in two-dimensions. There is an algorithm with sub-linear time, which is better than Bird's.

Example. row 1 = row 4, row 3 = row 5

Pattern	1 2 3 4 5	position	Vertical pattern
1	a a b b a	1	1
2	a a a b b	2	2
3	a b a b a	3	3
4	a a b b a	4	1
5	a b a b a	5	3

h function

	1 2 3 4 5
pat	1 2 3 1 3

1. Data structures for geometry

We deal with two-dimensional geometry. We express points by the struct type in C as follows:

```
struct point {  
    int x, y;  
}
```



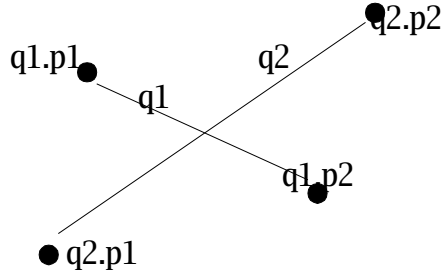
We express line segments by two points as follows:

```
struct segment {  
    struct point p1, p2;  
}
```



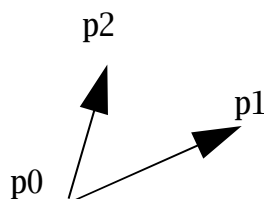
2. Intersection of two line segments

In this section, we develop an algorithm for determining whether two line segments intersect. Let two line segments be q_1 and q_2 . That is, the two end points of q_1 are $q_1.p_1$ and $q_1.p_2$, and those of q_2 are $q_2.p_1$ and $q_2.p_2$. We use the following condition for intersection. Note that a line segment defines an infinite (straight) line



Condition: The two end points $q_1.p_1$ and $q_1.p_2$ must be on the opposite sides of the line defined by q_2 , and the two end points $q_2.p_1$ and $q_2.p_2$ must be on the opposite sides of the line defined by q_1 .

As an auxiliary function, we make the function “turn” that tests whether the angle made by three points p_0 , p_1 , and p_2 . The function tests whether the vector $p_0 \rightarrow p_2$ is ahead of the vector $p_0 \rightarrow p_1$ anti-clockwise. If it is true, it returns 1, and -1 otherwise. See the following figure. The function value is 1 in this case.



The function “intersect” together with “turn” is given in the following.

```
struct point { int x, y; }
```

```
struct segment { struct point p1, p2;}
```

```
int turn {struct point p0, struct point p1, struct point p2)
```

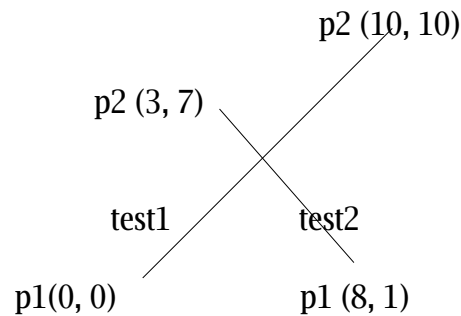
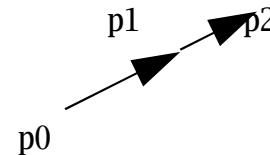
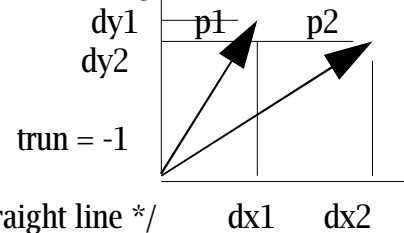
```
{ /* The last three cases are when three points are on a straight line */
  int dx1, dx2, dy1, dy2;
  dx1 = p1.x - p0.x; dy1 = p1.y - p0.y;
  dx2 = p2.x - p0.x; dy2 = p2.y - p0.y;
  if (dx1*dy2 > dy1*dx2) return 1;
  if ( dx1*dy2 < dy1*dx2) return -1;
  if ((dx1*dx2 < 0)|| (dy1*dy2 < 0)) return -1;
  if ((dx1*dx1 + dy1*dy1) < (dx2*dx2 + dy2*dy2)) return 1;
  return 0;
}
```

```
int intersect(struct segment s1, struct segment s2)
```

```
{ if(((s1.p1.x==s1.p2.x)&&(s1.p1.y==s1.p2.y))||
  ((s2.p1.x==s2.p2.x)&&(s2.p2.x==s2.p2.x)))
  /* case 1: One of two segments shrinks to one point */
  return ((turn(s1.p1, s1.p2, s2.p1)*turn(s1.p1,s1.p2,s2.p2))<=0)
  || ((turn(s2.p1, s2.p2, s1.p1)*turn(s2.p1,s2.p2,s1.p2))<=0);
  else /* case 2: Neither of segments is a single point */
  return ((turn(s1.p1, s1.p2, s2.p1)*(turn(s1.p1, s1.p2, s2.p2)) <= 0)
  &&((turn(s2.p1, s2.p2, s1.p1)*trun(s2.p1, s2.p2, s1.p2))) <= 0);
}
```

```
main() /* for test */
```

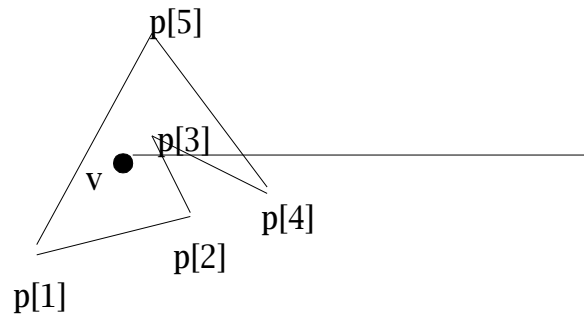
```
{
  struct segment test1, test2;
  test1.p1.x = 0; test1.p1.y = 0;
  test1.p2.x = 10; test1.p2.y = 10;
  test2.p1.x = 8; test2.p1.y = 1;
  test2.p2.x = 3; test2.p2.y = 7;
  printf(“ %d\n”, intersect(test1, test2));
}
```



This program correctly computes whether two segments intersect even in special cases like one point is on the other segment or one end point of one segment and one point of the other segment fall on the same point .

3. Interior point

We test if a point is inside a polygon. The polygon is not necessarily convex. We draw a semi-infinite line from the given point. If it intersects with the edges of the polygon an odd number of times, it is inside, and outside if even. When we traverse vertices of the polygon, we check if they are on the semi-infinite line. If so we skip them and a both-infinite line is used for intersection test. See the following figure.



This program is easy to design if we use the function “intersect”. We give the polygon by array p[1], ..., p[n] of points. The program follows.

```
int inside(struct point p[ ], int n, struct point v)
{
    /* y-coordinate of p[1] is the smallest. */
    int i, j = 0, count = 0;
    struct segment sw, sp;
    p[0] = p[n];
    sw.p1 = v; sw.p2.x = 20000; sw.p2.y = v.y; /* sw is the semi-infinite segment */
    ss.p1.y=v.y; ss.p2.y = v.y; ss.p1.x = -20000; ss.p2.x = 20000; /* ss is both-infinite */
    for (i=1; i<=n; i++) {
        sp.p1 = p[i]; sp.p2 = p[i];
        if (!intersect(sp, sw)) { /* if p[i] is on sw we skip */
            sp.p2 = p[j];
            if (i==j+1)
                if (intersect(sp, sw) count++; /* sp is segment from p[i] to p[i-1] */
            else if (intersect(sp, ss)) count++;
            j = i; /* j goes one step behind i */
        }
    }
    return count & 1 /* return 1 if count is odd, 0 otherwise */
}
main() /* for test */
```

```

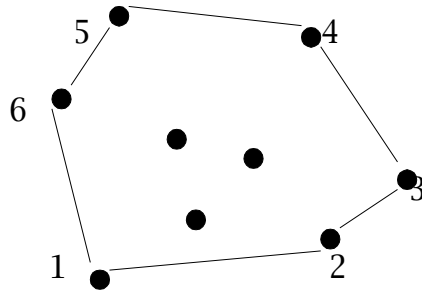
{
    struct point v;
    struct point p[10];
    int n = 5;
    p[1].x = 0; p[1].y = 0;
    p[2].x = 5; p[2].y = 1;
    p[3].x = 4; p[3].y = 5;
    p[4].x = 9; p[4].y = 2;
    p[5].x = 2; p[5].y = 8;
    v.x = 3; v.y = 3;
    printf(" %d\n", inside(p, n, v));
}

```

4. Finding the Convex Hull

A convex polygon is a polygon such that the inner angle at each vertex is less than 180 degrees. The convex hull of n points is the smallest convex polygon that includes those points. We first describe the package wrapping method with $O(n^2)$ time and then Graham's algorithm with $O(n \log n)$ time.

Package wrapping: We start from a specified point and select the point we first encounter by going anti-clockwise, and repeat this process by going to the selected point. In the following figure we start from point 1 and select point 2, ..., 6. Finally we select 1 and finish.



We need to compare the angles between segments. The C library function `arctan` will give us the angle in radian. If we just determine which of two angles is greater, we can use the following function “angle”, which is more efficient.

```

double angle(struct point q1, struct point q2)
{
    /* gives angle between 0 and 360 degrees */
    int dx, dy;
    double a = 0.0;
}

```

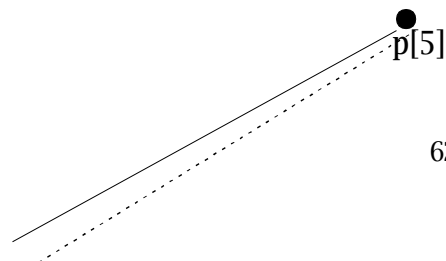
```

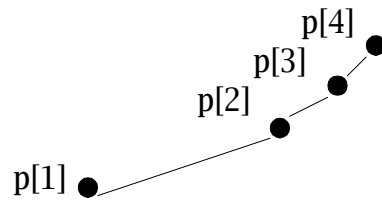
    dx = q2.x - q1.x; dy = q2.y - q1.y;
    if (dx != 0 || dy != 0) a = (double)dy / (double)(abs(dx) + abs(dy));
    if (dx < 0) a = 2.0 - a;
    else if (dy < 0) a = a + 4.0;
    return a*90.0;
}
int convex_hull(struct point p[ ], int n)
{
    /* p[1] through p[n] are vertices of a polygon */
    int i, m, min; double theta, prev;
    struct point t;
    min = 1;
    for (i=2; i<=n; i++) if (p[i].y<p[min].y) min = i;
    t = p[1]; p[1] = p[min]; p[min] = t;
    p[n+1] = p[1];
    min = n; theta = 360.0;
    for (i=2; i<=n; i++)
        if (angle(p[1], p[i]) < theta) {min = i; theta = angle(p[1], p[min]);}
    for (m=2; m<=n; m++) {
        t = p[m]; p[m] = p[min]; p[min] = t; /* swap p[m] and p[min] */
        prev = theta;
        min = n + 1; theta = 360.0;
        /*** This part finds the point p[min] such that angle(p[m], p[min]) is minimum
        and greater than the previous angle. ***/
        for (i=m+1; i<=n+1; i++)
            if ((angle(p[m], p[i]) < theta) && (angle(p[m], p[i]) >= prev))
                {min = i; theta = angle(p[m], p[min]);}
        if (min == n+1) return m; /* m is the number of points on the hull */
    }
}

```

As the program has a double nested loop structure, the computing time is $O(n^2)$, which is not very efficient. In the following, we introduce Graham's algorithm invented by R. L. Graham in 1972, which runs in $O(n \log n)$ time.

Graham's algorithm: We start from the same point as before, that is, the lowest point $p[1]$. We sort the angles $\text{angle}(p[1], p[i])$ for $i=1, \dots, n$ in increasing order by any fast sorting algorithm, such as quicksort. Let $p[1], \dots, p[n]$ be sorted in this way. Then we pick up $p[2], p[3], \dots$ in this order. We keep selecting points as long as we go anti-clockwise from the previous angle. If we hit a clockwise turn, we throw away already selected points until we recover clockwise angle. See the following figure.





After we select up to p[4], we throw away p[3] and p[4], and connect p[2] and p[5]. To determine clockwise turn or anti-clockwise turn, we use the function 'turn', introduced before. The main part of the algorithm follows.

```

struct point { int x, y; double angle; };
struct point a[100000];
int convex_hull(struct point p[], int n)
{
    /* p[1] through p[n] are vertices of a polygon */
    int i, m, min; double theta, prev;
    struct point t;
    min = 1;
    for (i=2; i<=n; i++) if (p[i].y < p[min].y) min = i;
    for (i=1; i<=n; i++) if (p[i].y==p[min].y)
        if (p[i].x > p[min].x) min = i;
    t = p[1]; p[1] = p[min]; p[min] = t;
    for (i=1; i<=n; i++) p[i].angle=angle(p[1],p[i]);
    for (i=1; i<=n; i++) a[i]=p[i];
    quick(1, n); /* This sorts a[] using keys a[i].angle for i=1, ..., n */
    for (i=1; i<=n; i++) p[i]=a[i];
    p[0] = p[n];
    m = 3;
    for (i=4; i<=n; i++) {
        while (turn(p[m], p[m-1], p[i]) >= 0) m--; /* throw away points */
        m++;
        t = p[m]; p[m] = p[i]; p[i] = t;
    }
    return m; /* m is the number of points on the hull */
} /* p[1], ..., p[m] are points on the hull */

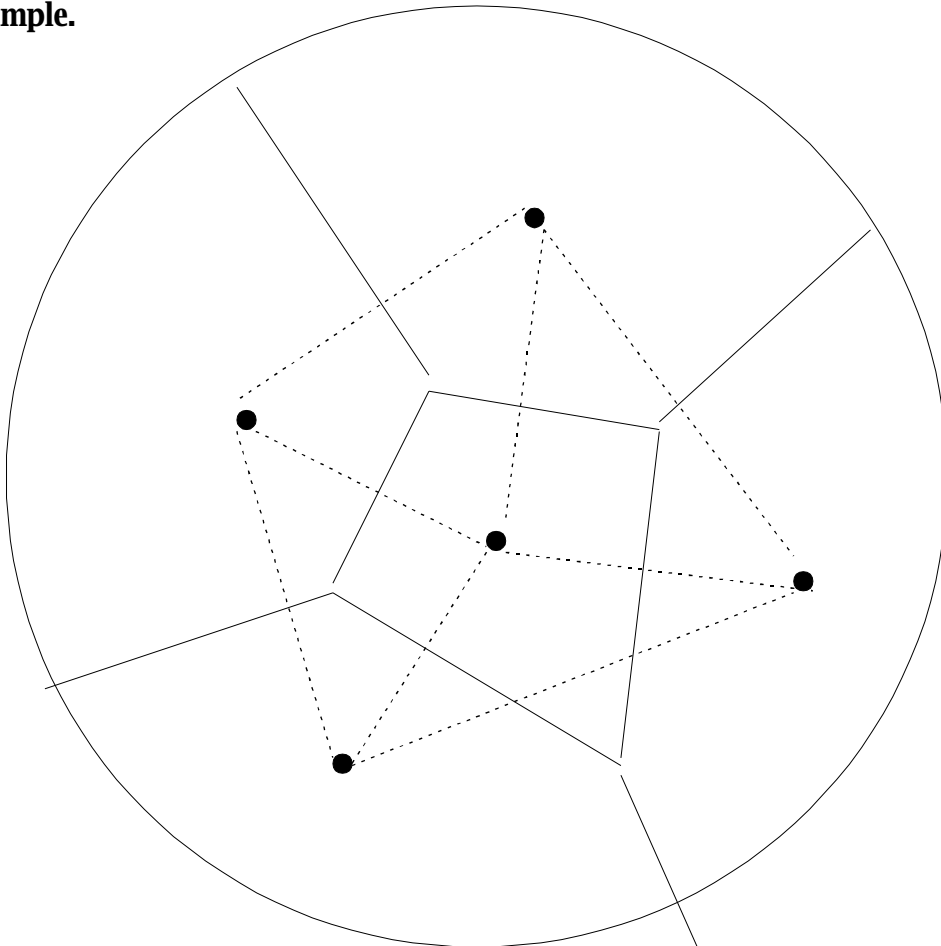
```

The main complexity is that of sorting. The time for the rest is $O(n)$ as we examine each point once. Hence the total time is $O(n \log n)$.

5. Voronoi Diagram

Suppose we have n points on a plane such as a square. The Voronoi diagram separates those points by line segments or semi-infinite lines in such a way that in any closed region there is only one point and any point in the region is closer to the point than to any other point. If we imagine a big circle surrounding those points, we can define a region with the semi-infinite lines and the circle. The regions define territories for those points, so to speak.

Example.



Note that the three perpendiculars of a triangle meet at one point. In this example five points are surrounded by line segments and the outer circle, and the five regions are defined. The line segment between two points, say, a and b, is the perpendicular of the points a and b. Any pair of two points which are in adjacent regions are connected by a dotted line. Those dotted lines define triangles, and the resulting diagram is called the Delauney diagram.

There are a number of algorithms for Voronoi diagrams. The following is based on the divide-and-conquer paradigm.

Algorithm. Voronoi Diagram

1. Sort the n points according to the x-co-ordinates and call the $p[1], \dots, p[n]$.
2. Call $\text{Voronoi}(1, n, P, V)$.
- 3 Define $\text{Voronoi}(l, r, P, V)$ as follows:
4. If there are only three or less in P , draw the diagram directly.
5. Let $m = (l+r)/2$.
6. Let $P_1 = \{p[l], \dots, p[m]\}$.
7. Call $\text{Voronoi}(l, m, P_1, V_1)$.
8. Let $P_2 = \{p[m+1], \dots, p[r]\}$.
9. Call $\text{Voronoi}(m, r, P_2, V_2)$.
10. Obtain the zipping line Z for V_1 and V_2 by algorithm M
11. Delete portions of line segments in V_1 exceeding Z to the right.
12. Delete portions of line segments in V_2 exceeding Z to the left.
13. Return the resulting diagram as V

Algorithm M

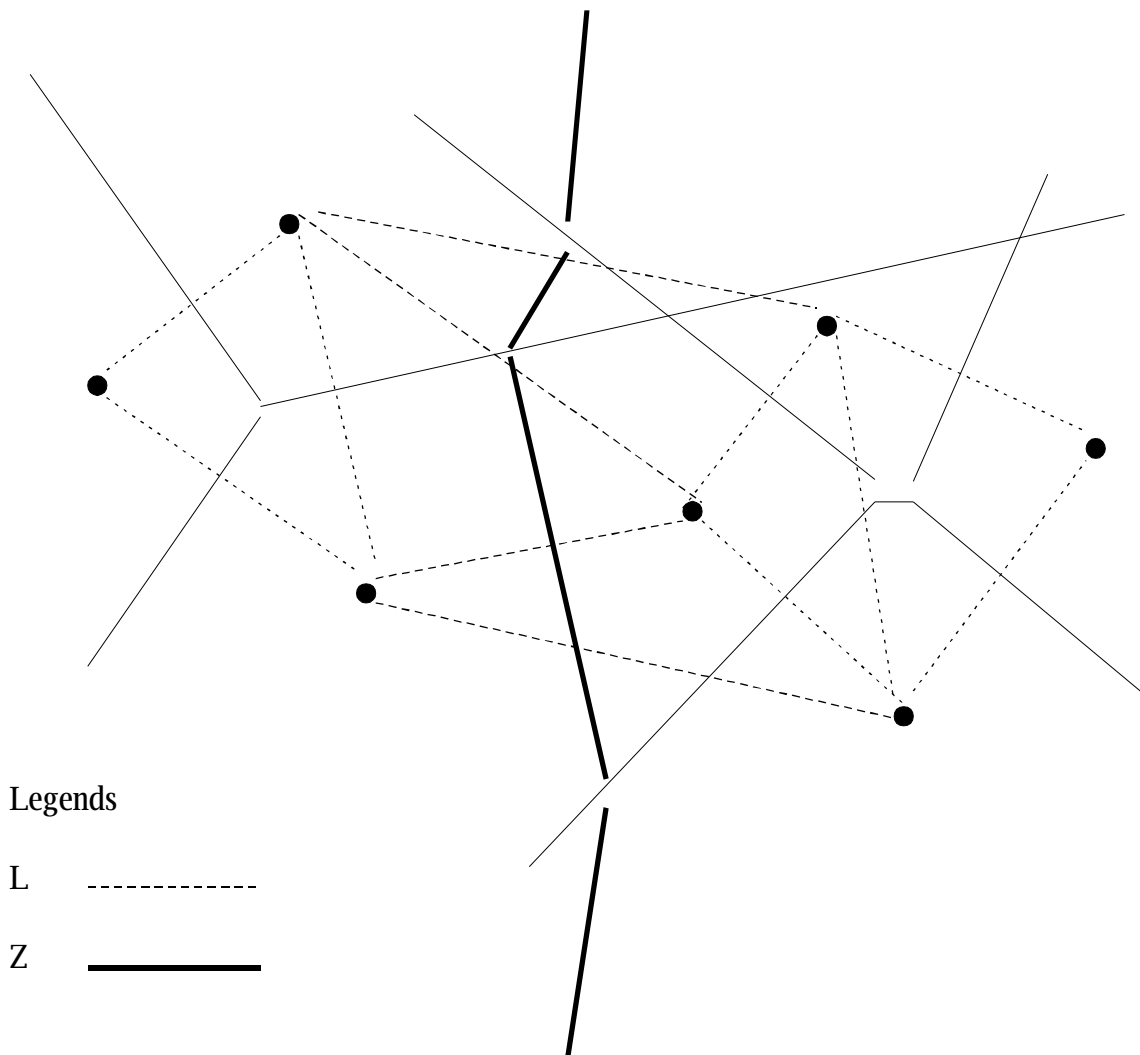
1. Obtain the convex hull for the combined set of P_1 and P_2
2. Let $L = (a, b)$ be the lower segment that connect a point in P_1 and a point in P_2 in the hull.
3. Originate Z as a point at the bottom.
4. Draw a perpendicular Z for L upwards until hitting a segment S in V_1 or V_2 .
5. If there is no such intersection, halt.
6. If S is in V_1 , it is a perpendicular for a and x for some x . Let a be x and go to 3.
7. If S is in V_2 , it is a perpendicular for b and y for some y . Let b be y and go to 3.

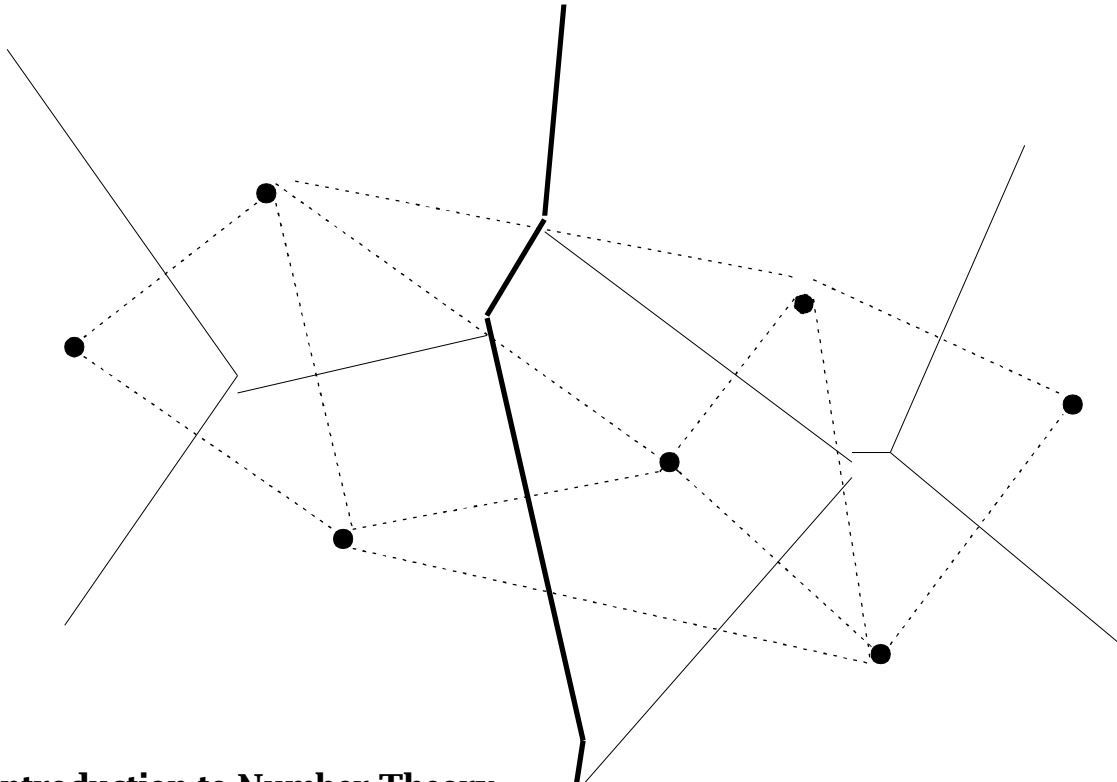
The time for Algorithm M is $O(n)$. As usual, we can establish the following recurrence for the computing time.

$$T(3) = c_1$$
$$T(n) = 2T(n) + c_2n$$

The solution is $T(n) = O(n \log n)$. Together with the sorting, the total time is $O(n \log n)$.

Example.





Introduction to Number Theory

Basics

Let Z be the set of integers and N be the set of natural numbers. That is,

$$Z = \{ \dots, -2, -1, 0, 1, 2, \dots \}$$

$$N = \{ 0, 1, 2, \dots \}$$

In the following, variables take integers by default.

Theorem. For any integer a and any integer $b > 0$, there are unique q and r such that

$$a = qb + r, 0 \leq r < b$$

Here q is the quotient and r is the remainder. If $r = 0$, that is, if b divides a , we write $b \mid a$.

If $m \mid a-b$ for $m > 0$, we write $a \equiv b \pmod{m}$, and say a is congruent to b modulo m .

Lemma. For fixed m , the relation " $\equiv \pmod{m}$ " is an equivalence relation, that is,

(1) reflexive, $a \equiv a \pmod{m}$

(2) symmetric, $a \equiv b \pmod{m} \Rightarrow b \equiv a \pmod{m}$

(3) transitive, $a \equiv b \pmod{m} \wedge b \equiv c \pmod{m} \Rightarrow a \equiv c \pmod{m}$.

By this lemma, we can partition Z by the relation " $\equiv \pmod{m}$ ", or simply " $\equiv (m)$ ", into equivalence classes. Let $[a]$ be the equivalence class including a , that is

$$[a] = \{ b \mid b = a + km, k \in Z \}.$$

Let the operation on equivalence classes be defined by

$$[a] \pm [b] = [a \pm b]$$

$$[a] * [b] = [a*b].$$

These operations are well defined thank to the following lemma. Then the quotient algebra $Z/\equiv (m)$ forms a finite ring of order m . We express $Z/\equiv (m)$ by representatives $\{0, 1, \dots, m-1\}$. We also denote $Z/\equiv (m)$ by Z_m .

Example. Z_5 .

$+$	0, 1, 2, 3, 4	$*$	0 1 2 3 4
0	0 1 2 3 4	0	0 0 0 0 0
1	1 2 3 4 0	1	0 1 2 3 4
2	2 3 4 0 1	2	0 2 4 1 3
3	3 4 0 1 2	3	0 3 1 4 2
4	4 0 1 2 3	4	0 4 3 2 1

Lemma. Let $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$. Then

$$a \pm c \equiv b \pm d \pmod{m}, a * c \equiv b * d \pmod{m}.$$

Let $\gcd(a, b)$ be the greatest common divisor of $a > 0$ and $b > 0$. If $\gcd(a, b) = 1$, a and b are said to be mutually prime.

Lemma. For any $a > 0$ and $b > 0$, there exist x and y such that $d = ax + by$ where $d = \gcd(a, b)$.

From this we have,

Lemma. For $a > 0$ and $m > 0$, $\gcd(a, m) = 1$ if and only if for some x , $a * x \equiv 1 \pmod{m}$.

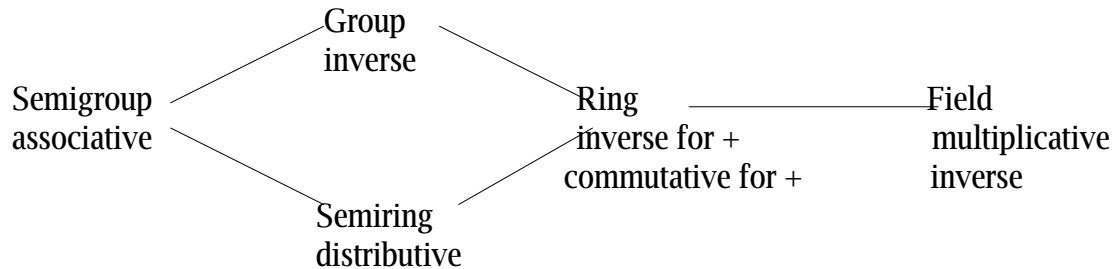
Corollary. Let $p > 0$ be a prime number. Then $\mathbb{Z}/\equiv (p)$ or \mathbb{Z}_p is a field.

Note. In a field F , any $a \in F$ has a multiplicative inverse a^{-1} , that is, $a * a^{-1} = 1$.

For algebraic systems, we have the following axioms.

Associative	$a(bc) = (ab)c$
Commutative	$ab = ba$
Unit element	$1a = a1 = a$
Inverse	$a a^{-1} = a^{-1} a = 1$
Distributive	$a(b+c) = ab + ac, (a+b)c = ac + bc.$

We have weaker systems to a stronger systems from left to right with additional axioms.



For $m > 0$, the number of integers between 1 and m which are mutually prime to m is called the Euler (totient) function, denoted by $\phi(m)$. That is,

$$\phi(m) = \{ a \in \mathbb{Z}_m \mid \gcd(a, m) = 1 \}.$$

$$\phi(1) = 1$$

$$\phi(2) = 1$$

$$\phi(3) = 2, \{1, 2\}$$

$$\phi(4) = 2, \{1, 3\}$$

$$\phi(5) = 4, \{1, 2, 3, 4\}$$

$$\phi(6) = 2, \{1, 5\}.$$

Let \mathbb{Z}_m^* be the set defined by

$$\mathbb{Z}_m^* = \{ a \in \mathbb{Z}_m \mid \gcd(a, m) = 1 \}.$$

Then the size of this set is $\phi(m)$. We have a few theorems.

Theorem. \mathbb{Z}_m^* forms a group under multiplication. The unit is 1.

Theorem. For any finite group such that the order is n , $x^n = 1$.

Proof. For $x \in G$, we generate $1, x, x^2, \dots$. Since G is finite we must have $x^a = x^b$ for some a and b such that $1 \leq a < b \leq n$. Then for $m = b - a$ such that $1 \leq m \leq n-1$, $x^m = 1$. The subset $\{1, x, \dots, x^{m-1}\}$ forms a subgroup of G . According to Lagrange's theorem, $|H|$ divides $|G|$, that is $m \mid n$. Thus $x^n = x^{km} = 1$ for some k .

Definition. Let H be a subgroup of group G . The right coset Ha of H in G specified by a is defined by

$$Ha = \{ ha \mid h \in H \}.$$

Congruence modulo H is defined by

$$a \equiv b \pmod{H} \Leftrightarrow ab^{-1} \in H.$$

Theorem. Congruence modulo H is an equivalence relation.

Lemma. $|H| = |Ha|$.

Proof. The mapping $f : H \rightarrow Ha$ defined by $f(h) = ha$ is a bijection (one-to-one and onto mapping).

Definition. The number of equivalence classes is the index denoted by $[G:H]$.

From this we see that $|G| = |H|[G:H]$. Hence,

Theorem (Lagrange). For a group and its subgroup H , $|H| \mid |G|$.

Since $|Z_m^*| = \phi(m)$, and for any $a \in Z_m^*$, $a^{\phi(m)} = 1$, we have the following.

Theorem (Fermat). For any $a \in Z$ and $m > 0$, if $\gcd(a, m) = 1$,

$$a^{\phi(m)} \equiv 1 \pmod{m}.$$

When $m = p$ is a prime in particular, and $\gcd(a, p) = 1$,

$$a^{p-1} \equiv 1 \pmod{p}.$$

Lemma. For $m > 0$, let the factoring of m be given by

$$m = (p_1^{e(1)})(p_2^{e(2)}) \dots (p_r^{e(r)}).$$

Then

$$\begin{aligned}\varphi(m) &= \varphi(p_1^{e(1)})\varphi(p_2^{e(2)})\dots\varphi(p_r^{e(r)}) \\ &= m(1-1/p_1)(1-1/p_2)\dots(1-1/p_r).\end{aligned}$$

In particular

$$\varphi(p^e) = p^{(e-1)}(p-1), \quad \varphi(2p^e) = p^{(e-1)}(p-1).$$

Definition. For mutually prime a and $m > 0$, and $e > 0$, let $a^e \equiv 1 \pmod{m}$ and for any e' such that $0 < e' < e$, $\text{not}(a^{e'} \equiv 1 \pmod{m})$. Then e is said to be the order of a modulo m .

Lemma. Let e be the order of $a \pmod{m}$ for $m > 0$. If $a^f \equiv 1 \pmod{m}$ for $f > 0$, then $e \mid f$.
Proof. There are q and r such that

$$f = qe + r, \text{ and } 0 \leq r < e.$$

If $r \neq 0$, we have

$$a^f \equiv a^{(qe+r)} \equiv a^{qe} a^r \equiv (a^e)^q a^r \equiv a^r \pmod{m}.$$

From this we have $a^r \equiv 1 \pmod{m}$, which contradicts the way we defined e .

In general, Z_m^* is a finite commutative group. If it is a cyclic group in addition, that is, for some a ,

$$Z_m^* = \{1, a, \dots, a^{\varphi(m)-1}\},$$

then a is said to be a primitive root of m . In other words, a is a primitive root of m if the order of a modulo m is $\varphi(m)$. Further in other words, a is a primitive root of m if

- (1) $a^{\varphi(m)} \equiv 1 \pmod{m}$, and
- (2) for any r such that $0 < r < \varphi(m)$, $\text{not}(a^r \equiv 1)$.

Example. For 7, $\varphi(7) = 6$. We enumerate the powers of some integers.

- 1, 2, 4, 1. 2 is not a primitive root.
- 1, 3, 2, 6, 4, 5, 1. 3 is a primitive root.

For 8, $\varphi(8) = 4$, and there is no primitive root.

Theorem. There are primitive roots only for $m = 2, 4, p^e, 2p^e$ for prime $p > 2$.

Evaluation of x^n

In stead of performing $n-1$ multiplications, we can compute x^n in $2\lfloor \log n \rfloor$ multiplications.

```
n = (b(m-1)...b(1)b(0)) : binary expression of n, e.g., 5 = (101)
y:=1;
for i:=m-1 downto 0 do begin
  y:=y*y;
  if b(i)=1 then y:=y*x
end.
```

This method is called “repeated squaring”.

Theory of Fibonacci Numbers

The Fibonacci sequence is defined by

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

That is

$$x(0) = 0, \quad x(1) = 1,$$
$$x(n+2) = x(n+1) + x(n), \quad (n \geq 0).$$

A homogeneous difference equation is defined by

$$x(0)=b(0), \quad x(1)=b(1), \quad \dots, \quad x(k-1)=b(k-1),$$
$$x(n+k) = a(1)x(n+k-1) + \dots + a(k)x(n).$$

The general solution is given by

$$x(n) = c(1)r(1)^n + \dots + c(k)r(k)^n,$$

where $r(i)$ is the i -th root of the characteristic equation

$$r^k - a(1)r^{k-1} - \dots - a(k) = 0.$$

Here we assume that all the k roots of the characteristic equation are different. The constants $c(1), \dots, c(k)$ are determined by the k initial conditions.

For the Fibonacci sequence, the characteristic equation is

$$r^2 - r - 1 = 0,$$

the solution of which is given by

$$r(1) = (1 + \sqrt{5})/2, r(2) = (1 - \sqrt{5})/2.$$

Therefore the general solution becomes as follows.

$$x(n) = c(1)r(1)^n + c(2)r(2)^n.$$

From the initial condition, we have the following simultaneous equation.

$$\begin{aligned} c(1) + c(2) &= 0 \\ c(1)(1 + \sqrt{5})/2 + c(2)(1 - \sqrt{5})/2 &= 1. \end{aligned}$$

From this we have

$$c(1) = 1/\sqrt{5}, c(2) = -1/\sqrt{5}.$$

Thus the n-th Fibonacci number x(n) is given by

$$x(n) = (1/\sqrt{5})((1 + \sqrt{5})/2)^n - (1/\sqrt{5})((1 - \sqrt{5})/2)^n$$

The growth rate of x(n) is measured by the first term which is dominant, that is, an exponential function of $r(1) = 1.618..$

With a primitive approach it takes $O(n)$ time to compute x(n). The following algorithm computes x(n) in $O(\log n)$ time, which is based on repeated squaring. The difference equation is expressed by vectors and a matrix as

$$(x(0) \ x(1)) = (0, 1)$$

$$(x(n-1) \ x(n)) = (x(n-2) \ x(n-1)) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

Then we have

$$(x(n-1) \ x(n)) = (x(0) \ x(1)) \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{(n-1)}$$

Hence we can use the fast algorithm to evaluate x^n based on repeated squaring as applied to a matrix.

Euclidean Algorithm for Greatest Common Divisors

We compute the greatest common divisor of a and $b > 0$, $\gcd(a, b)$. Let us divide a by b , yielding the following equation for the quotient q and remainder r .

$$a = bq + r, \quad 0 \leq r < b$$

We observe that $\gcd(a, b) = \gcd(b, r)$. The important point here is that the problem of $\gcd(a, b)$ has been reduced to that of $\gcd(b, r)$. Then we can repeat this process as shown below.

Let $a = r(0)$, $b = r(1)$, $q = q(1)$ and $r = r(2)$ in the above. We repeat division as follows:

$$\begin{aligned} a &= b \cdot q(1) + r(2), & 0 \leq r(2) < b \\ b &= r(2) \cdot q(2) + r(3), & 0 \leq r(3) < r(2) \\ &\dots \\ r(i-1) &= r(i) \cdot q(i) + r(i+1), & 0 \leq r(i+1) < r(i) \\ &\dots \\ r(n) &= r(n) \cdot q(n) + r(n+1), & r(n+1) = 0 \end{aligned}$$

That is we finish at the n -th division.

Theorem. $\gcd(a, b) = r(n)$.

Proof. Theorem follows from

$$\gcd(a, b) = \gcd(b, r(2)) = \gcd(r(2), r(3)) = \dots = \gcd(r(n), r(n+1)) = \gcd(r(n), 0) = r(n).$$

Example. $a = 91$, $b = 35$.

$$\begin{aligned} 91 &= (35)2 + 21, & q(1) = 2, & r(2) = 21 \\ 35 &= (21)1 + 14, & q(2) = 1, & r(3) = 14 \\ 21 &= (14)1 + 7, & q(3) = 1, & r(4) = 7 \\ 14 &= (7)2 + 0, & q(4) = 2, & r(5) = 0 \\ \gcd(91, 35) &= 7. \end{aligned}$$

Sequences $\{a(i)\}$ and $\{b(i)\}$ are defined as follows:

$$\begin{aligned} a(0) &= 0, & a(1) &= 1, & a(i) &= a(i-2) - q(i-1)a(i-1) \\ b(0) &= 1, & b(1) &= 0, & b(i) &= b(i-2) - q(i-1)b(i-1). \end{aligned}$$

Theorem. For any $i > 0$, $a \cdot b(i) + b \cdot a(i) = r(i)$.

Proof. Induction on i .

$$\begin{aligned} i=0, & \quad a \cdot b(0) + b \cdot a(0) = a = r(0) \\ i=1, & \quad a \cdot b(1) + b \cdot a(1) = b = r(1) \end{aligned}$$

Assume theorem is true for up to i . Then

$$\begin{aligned} a^*b(i+1) + b^*a(i+1) &= a(b(i-1) - q(i)b(i)) + b(a(i-1) - q(i)a(i)) \\ &= a^*b(i-1) + b^*a(i-1) - q(i)(a^*b(i) + b^*a(i)) \\ &= r(i-1) - q(i)r(i) \\ &= r(i+1). \end{aligned}$$

Let $i = n$ in the theorem. Then we have

$$a^*b(n) + b^*a(n) = \gcd(a, b).$$

That is we can find an integer solution (x, y) that satisfies the equation

$$ax + by = \gcd(a, b).$$

Example. For the previous example of $a = 91$ and $b = 75$. We have

$$\begin{aligned} a(2) &= -2, & a(3) &= 3, & a(4) &= -5 \\ b(2) &= 1, & b(3) &= -1, & b(4) &= 2. \end{aligned}$$

From this we have the solution $(x=1, y=-5)$ for $91x + 35y = 7$.

In the case of $\gcd(a, b) = 1$, the solution for $ax + by = 1$ has the following meaning.

$$x = a^{(-1)} \bmod b, \quad y = b^{(-1)} \bmod a.$$

Example. If we divide both sides of the previous example, we have $13x + 5y = 1$. Since 13 and 5 are mutually prime, each has a multiplicative inverse modulo each other. For example, $5^{(-1)} = -5$. To get a positive multiplicative inverse, we have $-5 + 13 = 8$.

The following procedure computes the greatest common divisor of a and b together with the sequences $\{a(i)\}$ and $\{b(i)\}$.

1. $w0:=a; w1:=b;$
 2. $u0:=0; u1:=1;$
 3. $v0:=1; v1:=0;$
 4. while $w1 > 0$ do begin
 5. $q := w0 \text{ div } w1;$
 6. $w2 := w0 - q*w1; w0:=w1; w1:=w2;$
 7. $u2 := u0 - q*u1; u0:=u1; u1:=u2;$
 8. $v2 := v0 - q*v1; v0:=v1; v1:=v2$
 9. end
- { $w0 = \gcd(a, b), u0 = b^{(-1)} \bmod a, v0 = a^{(-1)} \bmod b$ }.

If u_0 or v_0 is negative at the end, we can add a or b respectively to get a positive inverse.

Analysis of Euclidean algorithm

Let $D(a, b)$ be the number of divisions performed in the algorithm. Let $f(n)$ be defined by

$$\begin{aligned} f(0) &= 1, f(1) = 2, \\ f(n) &= f(n-1) + f(n-2), \quad n \geq 2 \end{aligned}$$

That is, $f(n) = x_{n+2}$ where $x(n)$ is the n -th Fibonacci number. Thus

$$f(n) = (1/\sqrt{5})((1+\sqrt{5})/2)^{n+2} - (1/\sqrt{5})((1-\sqrt{5})/2)^{n+2}.$$

Lemma.

$$\text{For all } n \geq 0, \quad f(n+1) \leq 2 * f(n), \quad f(n) \geq ((1+\sqrt{5})/2)^n$$

Theorem.

$$\text{For } 0 \leq b \leq a \leq f(n), \quad D(a, b) \leq n \text{ for } n \geq 0.$$

Proof. Induction on n .

Basis. $n=1$. Since $f(1) = 2$, we classify a and b such that $0 \leq b \leq a \leq 2$ into a few cases.

When $b=0$, $D(a, b)=0$. When $b=1$, $D(a, b)=1$. When $a=b=2$, $D(a, b)=1$.

Induction step. Assume theorem is true for up to n , and assume $0 \leq b \leq a \leq f(n+1)$. Then we classify the situation into two cases.

(1) $b \leq f(n)$. When $b=0$, $D(a, b)=0$. Let $b > 0$. Since $0 \leq r(2) < b$, we have $D(a, r(2)) \leq n$ from the inductive hypothesis. Thus $D(a, b) = D(b, r(2)) + 1 \leq n+1$.

(2) $f(n) < b$. From lemma, we have $a < 2b$. Thus

$$r(2) = a \bmod b = a - b < f(n+1) - f(n) = f(n-1).$$

When $r(2) = 0$, $D(a, b) = 1$. When $r(2) > 0$, we have $0 \leq r(3) < r(2) < f(n-1)$. From the inductive hypothesis,

$$D(a, b) = D(b, r(2)) + 1 = D(r(2), r(3)) + 2 < n-1 + 2 = n+1.$$

Now let a and b be positive integers expressed by up to k bits. Then we have $0 \leq b \leq a \leq 2^k - 1$. Let $\phi = (1+\sqrt{5})/2$ and $n = \lceil k \log_{\phi} 2 \rceil = 1.44k$. Then from $\phi^k \geq 2^k$ and lemma, where ϕ is the first root of the characteristic equation, we have

$$0 \leq b \leq a < \phi^n \leq f(n).$$

From theorem, we have $D(a, b) \leq n$. That is, the number of divisions is not more than $1.44k$. Let the time for division of two k -bit numbers be $O(D(k))$. Then the Euclidean algorithm runs in $O(kD(k))$ time. If we use the simple $O(k^2)$ algorithm for division, the Euclidean algorithm runs in $O(k^3)$ time.

Quadratic residues and probabilistic primality test

Let $m > 0$ and a such that $\gcd(a, m) = 1$ be given. If equation $x^2 \equiv a \pmod{m}$ has an integer solution, we say a is a quadratic residue modulo m , and otherwise a is said to be a quadratic non-residue modulo m . From this definition, we have if $a \equiv b \pmod{m}$,

a is a quadratic residue modulo $m \Leftrightarrow b$ is a quadratic residue modulo m .

If c is an integer solution for $x^2 \equiv a \pmod{m}$, then any d such that $d \equiv c \pmod{m}$ is also a solution. Thus we can only consider the range $1, 2, \dots, m-1$ for residues and solutions.

Theorem A. Let $p > 2$ be a prime. Any $a \geq 0$ such that $\gcd(a, p) = 1$ is a quadratic residue modulo p if and only if $a^{(p-1)/2} \equiv 1 \pmod{p}$. If a such that $\gcd(a, p) = 1$ is a quadratic non-residue modulo p , then we have $a^{(p-1)/2} \equiv -1 \pmod{p}$.

Proof. For a such that $\gcd(a, p) = 1$, suppose a is a quadratic residue modulo p . For solution c for equation $x^2 \equiv a \pmod{p}$, obviously we have $\gcd(c, p) = 1$. Thus raising both sides of $a \equiv c^2 \pmod{p}$ to the $(p-1)/2$ -th power, we have $a^{(p-1)/2} \equiv c^{p-1} \equiv 1 \pmod{p}$ from Fermat's theorem.

Conversely suppose $a^{(p-1)/2} \equiv 1 \pmod{p}$. Since $\gcd(a, p) = 1$, we have a primitive root g such that $a \equiv g^r \pmod{p}$ for some r . By raising both sides to the $(p-1)/2$ -th power, we have $g^{r(p-1)/2} \equiv a^{(p-1)/2} \equiv 1 \pmod{p}$. Since the order of $g \pmod{p}$ is $p-1$, and from the property of the order, $p-1$ must divide $r(p-1)/2$. Thus $r/2$ must be an integer. That is, $r = 2k$ for some integer k . From this $g^{2k} \equiv a \pmod{p}$ and a is a quadratic residue modulo p .

The latter half of the theorem can be seen in the following way. From Fermat's theorem, $a^{p-1} \equiv 1 \pmod{p}$. By factoring, we have

$$(a^{(p-1)/2} - 1)(a^{(p-1)/2} + 1) \equiv 0 \pmod{p}.$$

From the former half of the theorem, if a is not a quadratic non-residue modulo p , $a^{(p-1)/2}$ can not be divided by p . Thus we must have $a^{(p-1)/2} + 1 \equiv 0 \pmod{p}$.

Theorem B. Let $p > 2$ be a prime. Let $\gcd(a, p) = \gcd(b, p) = 1$. If both of a and b are quadratic residues or non-residues modulo p , then ab is a quadratic residue. If one is a residue and the other is a non-residue, then ab is a non-residue.

Proof. If $a^{(p-1)/2} \equiv b^{(p-1)/2} \equiv 1 \pmod{p}$, or $a^{(p-1)/2} \equiv b^{(p-1)/2} \equiv -1 \pmod{p}$, then $(ab)^{(p-1)/2} \equiv 1 \pmod{p}$. Hence from the previous theorem, if both are residues or non-residues, ab is a residue. If $a^{(p-1)/2} \equiv 1 \pmod{p}$ and $b^{(p-1)/2} \equiv -1 \pmod{p}$,

$(ab)^{(p-1)/2} \equiv -1 \pmod{p}$, and hence ab is a non-residue.

For a prime $p > 2$, we define the Legendre symbol $L(a, p)$ by

$L(a, p) = 1$, if a is a quadratic residue modulo p
 $L(a, p) = -1$, if a is a non-residue.

Theorem A says that for a prime $p > 2$ and a such that $\gcd(a, p) = 1$,

$$L(a, p) \equiv a^{(p-1)/2} \pmod{p}.$$

From previous discussions and Theorem B, we have

$$a \equiv b \pmod{p} \Rightarrow L(a, p) = L(b, p)$$
$$L(ab, p) = L(a, p)L(b, p).$$

Next we define the Jacobi symbol $J(a, m)$ for an odd number $m > 2$ and a such that $\gcd(a, m) = 1$ by

$$J(a, m) = L(a, p_1)L(a, p_2)\dots L(a, p_n),$$

where $m = p_1 * p_2 * \dots * p_n$ is a factorization of m into primes p_1, p_2, \dots, p_n .

For odd m and $n > 2$ such that $\gcd(a, m) = \gcd(b, m) = \gcd(m, n) = 1$, we have:

(1) $J(1, m) = 1$

(2) $a \equiv b \pmod{m} \Rightarrow J(a, m) = J(b, m)$
 $J(a, m) = J(a \pmod{m}, m)$

(3) $J(ab, m) = J(a, m)J(b, m)$

(4) $J(2, m) = (-1)^{(m^2 - 1)/8}$

(5) $J(m, n) = J(n, m)(-1)^{(m-1)(n-1)/4}$.

Since the Legendre symbol is a special case of the Jacobi symbol, the above five properties hold for the Legendre symbol as well.

If $m > 2$ is a prime, we have $J(a, m) = L(a, m)$, and hence

$$J(a, m) \equiv a^{(m-1)/2} \pmod{m}.$$

If for an odd number $m > 2$ and some a ($0 < a < m$), the above formula does not hold, we can judge that m is not a prime. This is called the Euler criterion for non-primality. If the above formula holds, however, we can not judge that m is a prime.

Let us judge that m is a prime if the formula holds for a random a such that $0 < a < m$ and $\gcd(a, m) = 1$. If m is a prime, this judgement is correct. In other words, if the formula does not hold, we can judge that m is a composite. If the number of a 's such that the formula holds for a composite m is not greater than $q(m-1)$ for $0 < q < 1$, the probability of wrong judgement is not greater than q . Let us test the formula k times for random a 's. If the formula does not hold before the end of k tests, we halt and judge that m is a composite. Then the probability of wrong judgement when we declare m is a prime after k successful tests is not greater than q^k . Solovay and Strassen (1977) showed that $q \leq 1/2$ as shown in the following theorem.

Let m be an odd composite number. Let formulas (1) and (2) be defined for $0 < a < m$ by

- (1) $\gcd(a, m) = 1$
- (2) $\gcd(a, m) = 1$ and $a^{(m-1)/2} \equiv J(a, m) \pmod{m}$.

Let $G(m)$ be the set of a such that a satisfies (1). Let $H(m)$ be the set of a such that a satisfies (2).

Theorem (Solovay and Strassen). $H(m)$ is a proper subgroup of $G(m)$. Therefore $|H(m)| \leq |G(m)|/2$

Proof. See R. Solovay and V. Strassen, "A fast Monte-Carlo test for primality," SIAM Jour. on Computing, vol. 6, no. 1 (1977), pp. 84-85 and vol. 7, no. 1 (1978) p. 118.

Thus by the Euler criterion, we have $q = 1/2$ for probabilistic primality test. Now we proceed to an algorithm for computing formula (2).

To compute $a^{(m-1)/2} \pmod{m}$, we can use repeated squaring with mod m operation inserted after each multiplication. Thus, if we use a primitive $O(k^2)$ method for multiplication and division with $k = \lceil \log_2 m \rceil$ bits, we can compute the above in $O(k^3)$ time. Now $J(a, m)$ can be computed by the following algorithm.

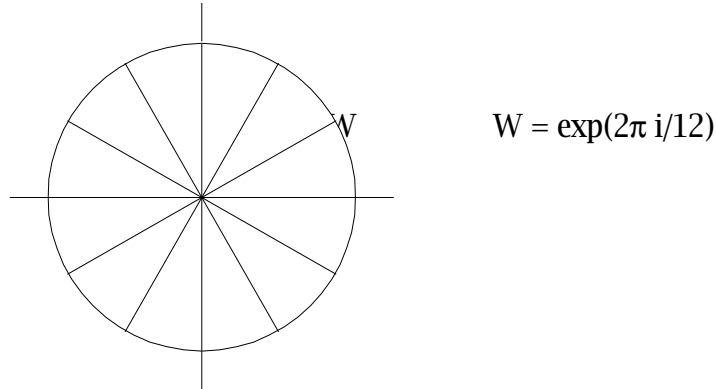
1. function $J(a, m)$
2. begin
3. if $m=0$ then report "a is composite"
4. else if $a=1$ then $J:=1$
5. else if a is even then $J := J(a/2, m) * (-1)^{((m-1)/8)}$
6. else $J := J(m \bmod a, a) * (-1)^{((a-1)(m-1)/4)}$
7. end

Line 6 is a part of the Euclidean algorithm, and hence the time for this algorithm is $O(k^3)$. Thus for one a , the Euler criterion takes $O(m^3)$ time.

Introduction to Fast Fourier Transform (FFT)

Basics

Let W be the principal n -th root of 1 in the complex plane, that is, $W = \exp(2\pi i/n)$, where $i = \sqrt{-1}$. See below.



Then we have

$$(1) \quad W^n = 1$$

$$(2) \quad \sum_{j=0}^{n-1} W^{kj} = 0 \quad (k = 0, \dots, n-1)$$

The discrete Fourier transform (DFT) of the sequence of complex numbers $X(j)$ ($j=0, \dots, n-1$) is defined by the sequence $y(k)$ ($k=0, \dots, n-1$) where

$$y(k) = \sum_{j=0}^{n-1} x(j)W^{kj}.$$

The inverse discrete Fourier transform (IDFT) of sequence $\{y(k)\}$ is defined by the sequence $z(l)$ where

$$z(l) = (1/n) \sum_{k=0}^{n-1} y(k)W^{-lk} \quad (l = 0, \dots, n-1)$$

Theorem. $x(j) = z(j)$ for $j = 0, \dots, n-1$. That is,

$$x(j) = (1/n) \sum_{k=0}^{n-1} y(k)W^{-kj}$$

Proof. Let $\mathbf{x} = (x(0), \dots, x(n-1))$, $\mathbf{y} = (y(0), \dots, y(n-1))$, and $\mathbf{z} = (z(0), \dots, z(n-1))$. Let matrices A and B be defined as $A = (a(i,j))$ and $B = (b(i,j))$ where

$$a(i,j) = W^{(ij)}, \quad b(i,j) = (1/n)W^{(-ij)}$$

Then we have $y = xA$ and $z = yB$. It suffices to show that $B = A^{(-1)}$. Let $C = BA$. Then we have

$$c(i,j) = (1/n) \sum_{[k=0 \text{ to } n-1]} W^{(j-i)k}$$

When $i = j$, $W^{(j-i)k} = 1$, so $c(i,i) = 1$. When $i \neq j$, let $l = j - i$. Then from (2), we have

$$\sum_{[k=0 \text{ to } n-1]} W^{(lk)} = 0.$$

Thus $c(i,j) = 0$ for $i \neq j$.

From this we see that IDFT is indeed an inverse transform.

The convolution of two sequences $(x(0), \dots, x(n-1))$ and $(y(0), \dots, y(n-1))$ is defined by $(z(0), \dots, z(2n-1))$ where

$$z(j) = \sum_{[k=0 \text{ to } n]} x(j-k)y(k) \quad (j = 0, \dots, 2n-1).$$

Here we assume $x(k) = 0$ for $k < 0$ and $k \geq n$. From this definition, it is clear that $z(2n-1) = 0$, which is added to the sequence to adjust to the length of $2n$.

Theorem. Let the DFTs of $(x(0), \dots, x(n-1), 0, \dots, 0)$ and $(y(0), \dots, y(n-1), 0, \dots, 0)$ of length $2n$ each be $(x'(0), \dots, x'(2n))$ and $(y'(0), \dots, y'(2n))$. The convolution $(z(0), \dots, z(2n))$ defined above is equal to the IDFT of $(x'(0)y'(0), \dots, x'(2n-1)y'(2n-1))$.

Proof. Let the DFT of $(z(0), \dots, z(2n-1))$ be $(z'(0), \dots, z'(2n-1))$. Then

$$\begin{aligned} z'(l) &= \sum_{[j=0 \text{ to } n-1]} \sum_{[k=0 \text{ to } n-1]} x(j-k)y(k)W^{(lj)} \\ &= \sum_{[k=0 \text{ to } n-1]} \sum_{[i-k \text{ to } 2n-1-k]} x(i)y(k)W^{(l(i+k))}, \quad \text{where } i = j-k \\ &= \sum_{[k=0 \text{ to } n-1]} \sum_{[j=0 \text{ to } n-1]} x(i)y(k)W^{(l(i+k))} \\ &= \sum_{[i=0 \text{ to } n-1]} x(i)W^{(li)} \sum_{[k=0 \text{ to } n-1]} y(k)W^{(lk)}. \end{aligned}$$

On the other hand, for $(x'(0), \dots, x'(2n-1))$ and $(y'(0), \dots, y'(2n-1))$ we have

$$x'(l) = \sum_{[k=0 \text{ to } 2n-1]} x(k)W^{lk} = \sum_{[k=0 \text{ to } n-1]} x(k)W^{lk} \quad (l = 0, \dots, 2n-1)$$

$$y'(l) = \sum_{[k=0 \text{ to } 2n-1]} y(k)W^{lk} = \sum_{[k=0 \text{ to } n-1]} y(k)W^{lk} \quad (l = 0, \dots, 2n-1)$$

Thus we have $z'(l) = x'(l)y'(l) \quad (l = 0, \dots, 2n-1)$.

Next let us consider multiplying two polynomials

$$\begin{aligned} f(z) &= x(0) + x(1)z + \dots + x(n-1)z^{n-1} \\ g(z) &= y(0) + y(1)z + \dots + y(n-1)z^{n-1}. \end{aligned}$$

The product is given by

$$\begin{aligned} f(z)g(z) &= x(0)y(0) + (x(0)y(1) + x(1)y(0))z \\ &\quad + \dots \\ &\quad + (x(0)y(k) + x(1)y(k-1) + \dots + x(k)y(0))z^k \\ &\quad + \dots \\ &\quad + x(n-1)y(n-1)z^{2n-2} + 0z^{2n-1}. \end{aligned}$$

That is, the k -th coefficient is the convolution $z(k)$. Next consider multiplying two multi-precision binary numbers $(a(n-1)\dots a(1)a(0))$ and $(b(n-1)\dots b(1)b(0))$. As we see from the following figure, we can modify the convolution computation for multiplication. Let the binary form of the product be $(c(2n-1)\dots c(1)c(0))$.

$$\begin{array}{rcccc} & & a(n-1) & \dots & a(1) & a(0) \\ & & b(n-1) & \dots & b(1) & b(0) \\ \hline & & a(n-1)b(0) & & a(1)b(0) & a(0)b(0) \\ & a(n-1)b(1) & & & a(0)b(1) & \\ \hline & a(n-1)b(n-1) & & a(0)b(n-1) & & \\ \hline c(2n-1) & c(2n-2) & & c(n-1) & c(1) & c(0) \end{array}$$

Here $c(i) \ (i=0, \dots, 2n-1)$ can be computed in the following way. Let $d(i) \ (i=0, \dots, 2n-1)$ be the convolution of $(a(0), \dots, a(n-1))$ and $(b(0), \dots, b(n-1))$.

```

c(0) := d(0);
for i:= 0 to 2n-1 do begin
  c(i+1) := d(i+1) + c(i) div 2;
  c(i) := c(i) mod 2
end.

```

If we go through FFT, all computations described in this section can be done in $O(n \log n)$ time, whereas straightforward methods take $O(n^2)$ time.