L31: Greedy Technique

Greedy design technique is primarily used in Optimization problems
The Greedy approach helps in constructing a solution for a problem through a sequence of steps (piece by piece) where each step is considered to be a partial solution. This partial solution is extended progressively to get the complete solution.
The choice of each step in a greedy approach is done based on the following
It must be feasible
It must be locally optimal
It must be irrevocable
A locally optimal choice is considered globally optimal.
Optimization problems are problems where in we would like to find the best of all possible solutions. In other words we need to find the solution which has the optimal (maximum or minimum) value satisfying the given constraints. In the greedy approach each step chosen has to satisfy the constraints given in the problem. Each step is chosen such that it is the best alternative among all feasible choices that are available. The choice of a step once made cannot be changed in subsequent steps.

Proof of correctness is necessary and important.
In general, greedy algorithms have five pillars:
A candidate set, from which a solution is created
A selection function, which chooses the best candidate to be added to the solution
A feasibility function that is used to determine if a candidate can be used to contribute to a solution
An objective function, which assigns a value to a solution, or a partial solution, and
A solution function, which will indicate when we have discovered a complete solution

Coin Change Problem

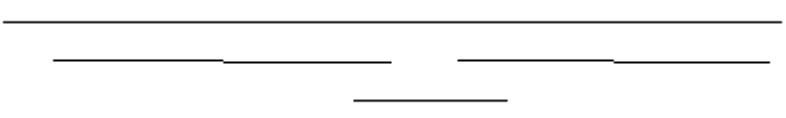
An amount to reach and an collection of coins to reach to that amount
Objective is to minimize the number of coins
Greedy theory says always return the largest coin
if coins are of denomination 32, 8, 1 it has the greedy choice property because no amount over 32 can be made without omitting 32.
if coins are of denomination 30,20,5,1 then it does not have greedy choice property because 40 is best made with two 20 coins but greedy will return 30,5,5 coins.

Interval problem(Activity Selection

Given set S = {a_1, …, a_n} of activities and activity start and finish times, e.g.:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Activities are compatible if their time intervals do not overlap, i.e.,
a_i is compatible with a_j is $s_i \geq f_j$ or $s_j \geq f_i$.
To find Subset of non overlapping intervals so that number of intervals are maximum.
First trial: Pick interval of smallest span



Second Trial: Pick the interval which overlaps with the smallest number of intervals and throw away those overlapping intervals

Interval that ends first and remove the intervals that overlap with these and recurse.

Activity selection problem

Given set S = {a_1, …, a_n} of activities and activity start and finish times, e.g.:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

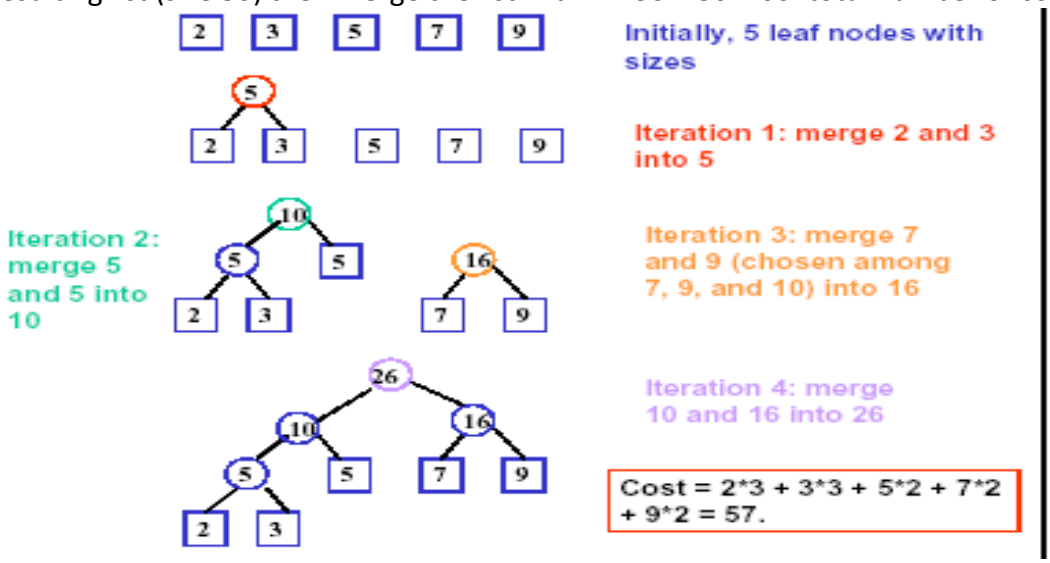
Activities are compatible if their time intervals do not overlap, i.e., a_i is compatible with a_j is $s_i \geq f_j$ or $s_j \geq f_i$.
Find a maximum-size subset of mutually compatible activities.
$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$
c[i,j] = number of activities in a max-size subset of mutually Compatible activities in S_{ij}

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n ← length[s]
2  A ← {a_1}
3  i ← 1
4  for m ← 2 to n
5       do if s_m ≥ f_i
6             then A ← A ∪ {a_m}
7                  i ← m
8  return A
```

Optimal Merge Patterns

Suppose there are 3 sorted lists L1, L2, and L3, of sizes 30, 20, and 10, respectively, which need to be merged into a combined sorted list but we can merge only two at a time.

We intend to find an optimal merge pattern which minimizes the total number of comparisons.
merge L1 & L2,: 30 + 20 = 50 comparisons , then merge the list & L3: 50 + 10 = 60 comparisons
total number of comparisons: 50 + 60 = 110. Alternatively, merge L2 & L3: 20 + 10 = 30 comparisons, the resulting list (size 30) then merge the list with L1: 30 + 30 = 60 total number of comparisons: 30 + 60 = 90.



Huffman Compression

File f has one million characters. only a,b,c,d,e,f chars come in the file then the total size of the file is 8 million bits

if we represent these with 000 001 010 011 101 111 then only 3 million bits
0 1 00 11 10 11 will give us less then 2 million bits
assume frequency of occurrence of a, b, c, d, e, f is 45%,13%,12%,16%,9%,5%
leaves of the binary tree will represent the bit strings
It will give us the prefix free code. use optimal merge patterns for making the binary tree and then assign 0 to every left node and 1 to every right node. The nodes with larger frequency will be nearer to the root and with lesser frequency will be towards the leaves

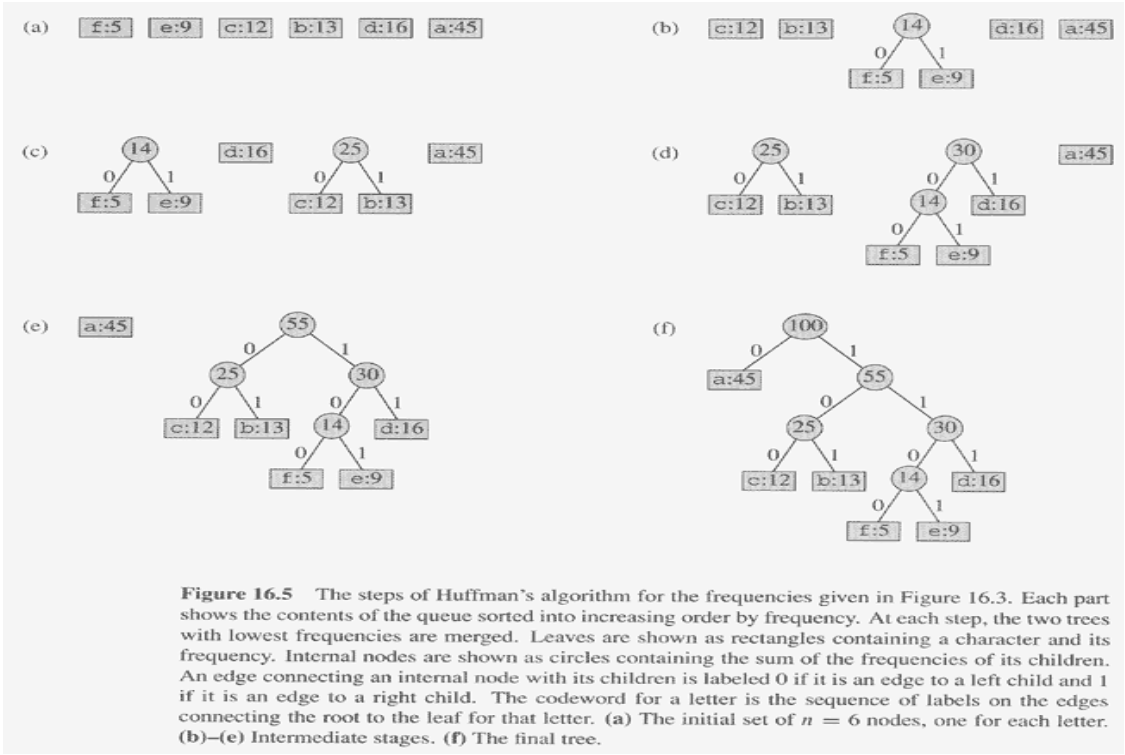
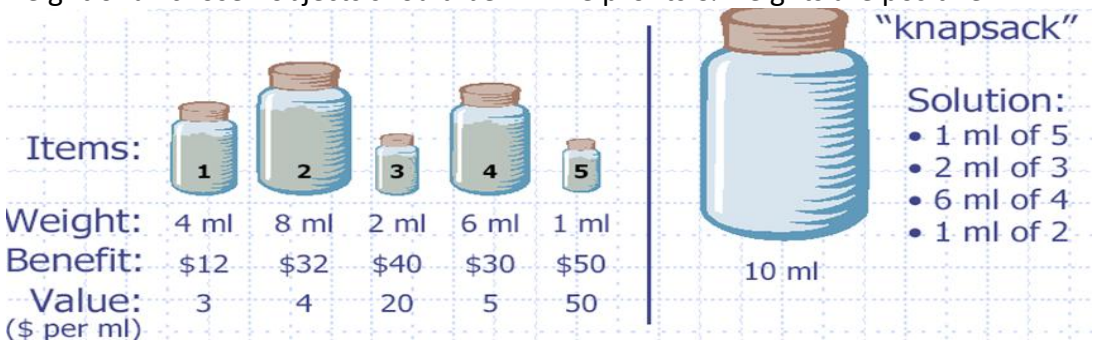


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of its children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. **(a)** The initial set of $n = 6$ nodes, one for each letter. **(b)–(e)** Intermediate stages. **(f)** The final tree.

a b c d e f
0 111 110 101 1001 1000
Total = 1*45+3*13+3*12+3*16+4*9+4*5=224
So total bits used are 2,24,000 bits

Partial Knapsack Problem

We are given n objects and a knapsack. Object i has weight wi and the knapsack has a capacity M. if a fraction xi, 0<= xi <= 1 of object i is placed into the knapsack then a profit of pixi is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is M, the total weight of all chosen objects should be M. The profits & weights are positive.



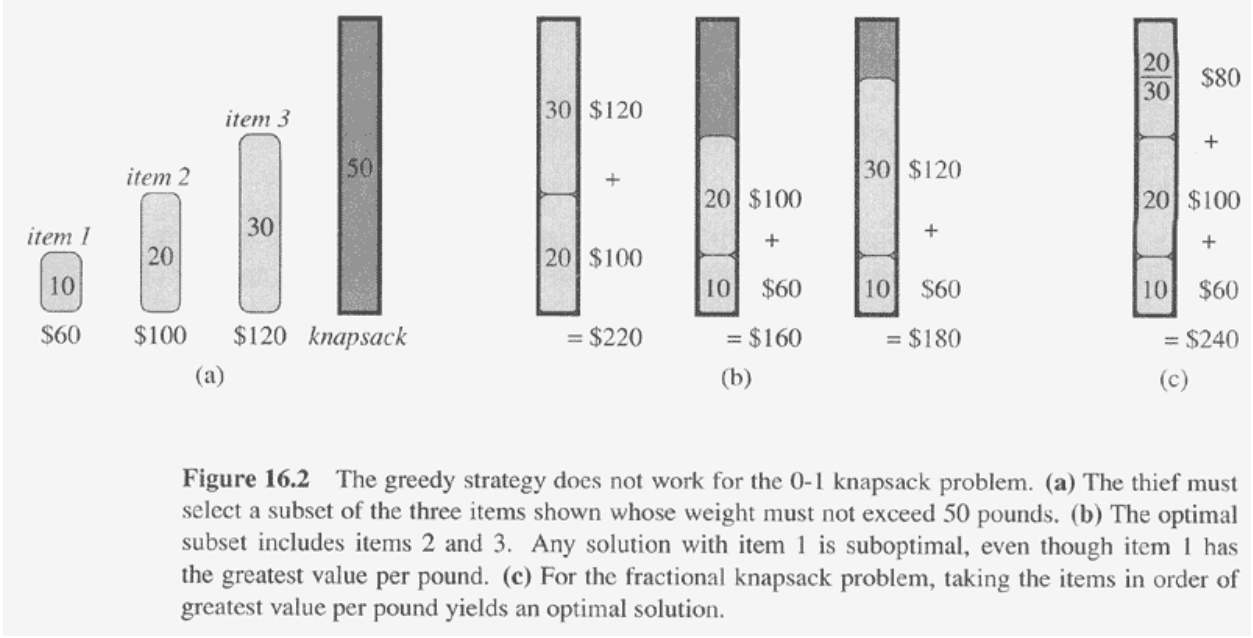


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Input Set S of items with weight w_i and benefit b_i with total weight W
Output Amount xi of each item i to maximize benefit with weight at most W
for each item i in S
xi=0
vi=bi/wi
w=0
while w<W
remove item i with highest vi
xi=min(wi,W-w)
w=w+min(wi,W-w)

Job Scheduling

Input: Set of jobs with processing time t_1, t_2,….t_n
There is no preemption and these have to be allocated processor. Only one processor is available. Output of the problem is the schedule of the jobs on the processor.
The finishing time of i_{th} job is $t_1+t_2+….t_i$
Objective is to minimize the sum of the finishing times
$f1=t_1$
$f2=t_1+t_2$
$fi=t_1+t_2+……+t_i$
:
$fn=nt_1+(n-1)t_2+……+t_n$
So t_1 should be smallest and so on.

L32 : Dynamic programming

The difference between Dynamic Programming and Divide and Conquer is that the sub problems in Divide and Conquer are considered to be disjoint and distinct but in Dynamic Programming they are overlapping.
Dynamic programming always gives a correct solution. It also deals with optimization problems and for most of these, the best known algorithm runs in exponential time.

We need to find a recursive formula for the solution. We can recursively solve sub problems, starting from the trivial case, and save their solutions in memory. In the end we'll get the solution of the whole problem
Optimal Substructure of the problems: The solution to the problem must be a composition of sub problem solutions
Sub problem Overlap : Optimal sub problems to unrelated problems can contain sub problems in common
Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
Simple Sub problems .We should be able to break the original problem to smaller sub problems that have the same structure

Matrix Chain Multiplication

We can multiply two matrices *A* and *B* only if they are ***compatible***: the number of columns of *A* must equal the number of rows of *B*. If *A* is a $p \times q$ matrix and *B* is a $q \times r$ matrix, the resulting matrix *C* is a $p \times r$ matrix. The time to compute *C* is dominated by the number of scalar multiplications which is *pqr*
Compute $A=A_0*A_1*\ldots*A_{n-1}$
A_i is $d_i \times d_{i+1}$
Problem: How to parenthesize?
Example B is 3×100 C is 100×5 D is 5×5
(B*C)*D takes 1500 + 75 = 1575 ops
B*(C*D) takes 1500 + 2500 = 4000 ops
Want to multiply matrices
A x B x C x D x E
We could parenthesize many ways
(A x (B x (C x (D x E))))
((((A x B) x C) x D) x E)…
Each different way presents different number of multiplies!
How do we figure out the wise approach?
Original matrix chain product
A x B x C x D x E (ABCDE for short)
Calculate in advance the cost (multiplies)
AB, BC, CD, DE
Use those to find the cheapest way to form
ABC, BCD, CDE
From that derive best way to form
ABCDE
Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
Calculate number of ops for each one
Pick the one that is best. Running time: The number of paranethesizations is equal to the number of binary trees with n nodes exponential

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

MATRIX-CHAIN-ORDER(p)

```
1 n ← length[p] - 1
2 for i ← 1 to n
3 do m[i, i] ← 0
4 for l ← 2 to n ▷ l is the chain length.
5 do for i ← 1 to n - l + 1
6 do j ← i + l - 1
7 m[i, j] ← ∞
8 for k ← i to j - 1
9 do q ← m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
10 if q < m[i, j]
11 then m[i, j] ← q
12 s[i, j] ← k
13 return m and s
```

```
PRINT-OPTIMAL-PARENS(s, i, j)
1 if i = j
2 then print "A"_i
3 else print "("
4 PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5 PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6 print ")"
```

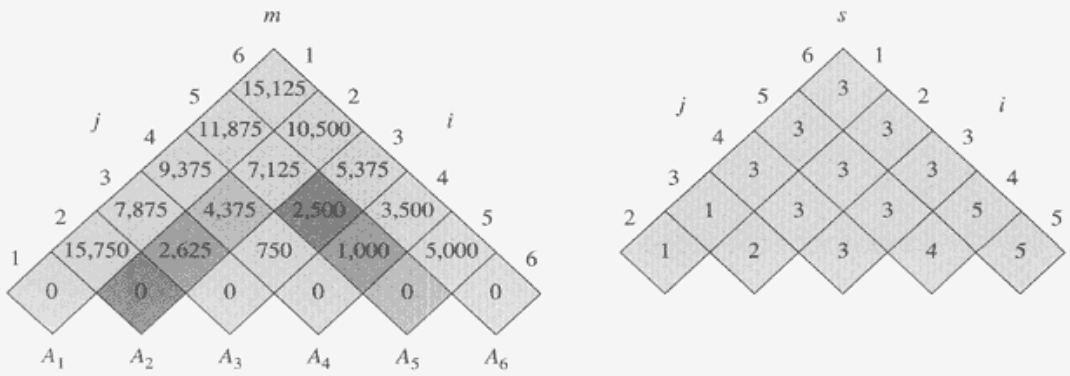


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$
$$= 7125\,.$$

0/1 knapsack using dynamic programming

S_k: Set of items numbered 1 to k.
Define B[k,w] = best selection from S_k with weight exactly equal to w
this does have subproblem optimality:
I.e., best subset of S_k with weight exactly w is either the best subset of S_{k-1} w/ weight w or the best subset of S_{k-1} w/ weight w-w_k plus item k.

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

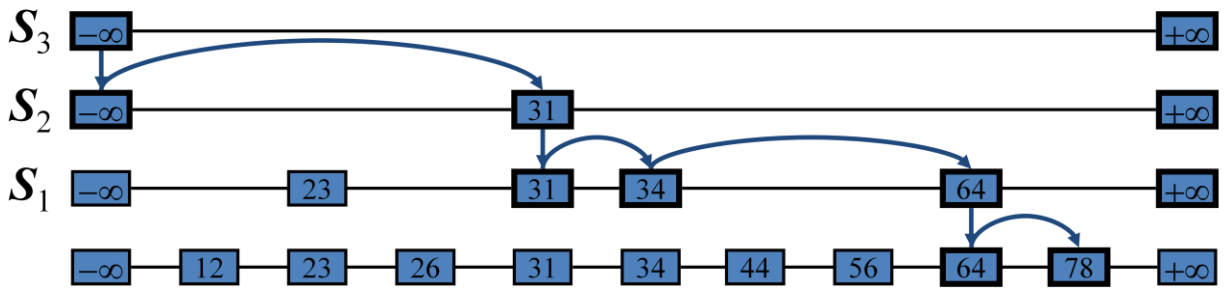
Running time: O(nW).
Not a polynomial-time algorithm if W is large
This is a pseudo-polynomial time algorithm
Algorithm *01Knapsack*(*S, W*):

Input: set *S* of items w/ benefit b_i and weight w_i; max. weight *W*
Output: benefit of best subset with weight at most *W*
for *w* ← 0 to *W* do
B[*w*] ← 0
for i = 1 to n
B[i,0] = 0
for *k* ← 1 to *n* do
for *w* ← *W* downto w_k do
if *B*[*w*-w_k]+b_k > *B*[*w*] then
B[*w*] ← *B*[*w*-w_k]+b_k

	0	1	2	3	4	5	6	7	8	9	Item:W,P
0	0	0	0	0	0	0	0	0	0	0	**1 : 4,20**
1	0	0	0	0	20	20	20	20	20	20	**2: 2,3**
2	0	0	3	3	20	20	23	23	23	23	**3 : 2,6**
3	0	0	6	6	20	20	26	26	29	29	**4 : 6,25**
4	0	0	6	6	20	20	26	26	31	31	**5 :2,80**
5	0	0	80	80	86	86	100	100	106	106	

L33: Skip List Data Structure

A skip list for a set S of distinct (key, element) items is a series of lists S0, S1 , … , Sh such that Each list Si contains the special keys +∞ and -∞. List S0 contains the keys of S in nondecreasing order Each list is a subsequence of the previous one, i.e., S0 ⊇ S1 ⊇ … ⊇ Sh
List Sh contains only the two special keys

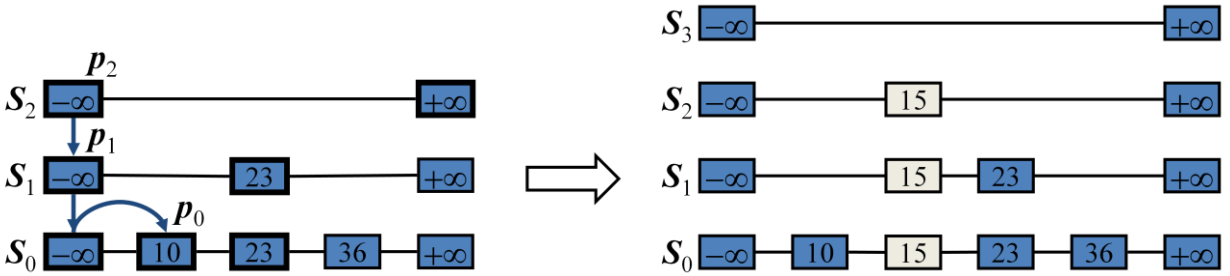


Searching

search for a key x in a a skip list as follows:
Start at the first position of the top list
At the current position p, we compare x with y ← key(after(p))
x = y: we return element(after(p))
x > y: we "scan forward"
x < y: we "drop down"
If we try to drop down past the bottom list, we return NO_SUCH_KEY
Example: search for 78

Insertion

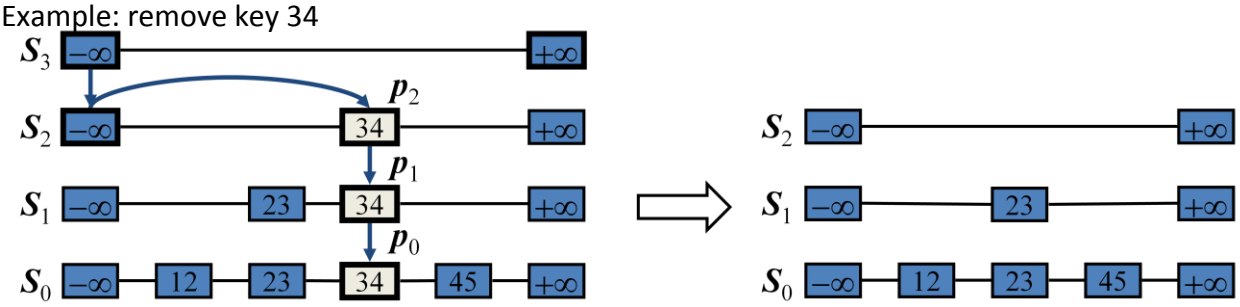
To insert an item (***x***, ***o***) into a skip list, we use a randomized algorithm:
We repeatedly toss a coin until we get tails, and we denote with ***i*** the number of times the coin came up heads
If $i \geq h$, we add to the skip list new lists S_{h+1}, … , S_{i+1}, each containing only the two special keys
We search for ***x*** in the skip list and find the positions p_0, p_1 , …, p_i of the items with largest key less than ***x*** in each list S_0, S_1, … , S_i
For $j \leftarrow 0, \ldots, i$, we insert item (***x***, ***o***) into list S_j after position p_j



Deletion

To remove an item with key ***x*** from a skip list, we proceed as follows:
We search for ***x*** in the skip list and find the positions p_0, p_1 , …, p_i of the items with key ***x***, where position p_j is in list S_j
We remove positions p_0, p_1 , …, p_i from the lists S_0, S_1, … , S_i

We remove all but one list containing only the two special keys
Example: remove key 34



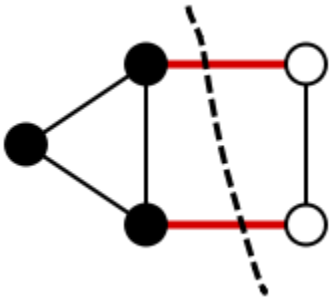
A skip list is a data structure for dictionaries that uses a randomized insertion algorithm
In a skip list with n items The expected space used is O(n)
The expected search, insertion and deletion time is O(log n)
The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm
We use the following two basic probabilistic facts:
Fact 1: The probability of getting i consecutive heads when flipping a coin is 1/2i
Fact 2: If each of n items is present in a set with probability p, the expected size of the set is np
Consider a skip list with n items. By Fact 1, we insert an item in list Si with probability 1/2i
By Fact 2, the expected size of list Si is n/2i . The expected number of nodes used by the skip list is

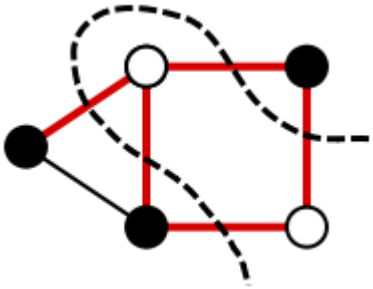
$$\sum_{i=0}^{h} \frac{n}{2^i} = n\sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

Thus, the expected space usage of a skip list with n items is $O(n)$
The running time of the search an insertion algorithms is affected by the height h of the skip list
We show that with high probability, a skip list with n items has height $O(\log n)$
We use the following additional probabilistic fact:
Fact 3: If each of n events has probability p, the probability that at least one event occurs is at most np
Consider a skip list with n items
By Fact 1, we insert an item in list S_i with probability $1/2^i$
By Fact 3, the probability that list S_i has at least one item is at most $n/2^i$
By picking $i = 3\log n$, we have that the probability that $S_{3\log n}$ has at least one item is at most
$n/2^{3\log n} = n/n^3 = 1/n^2$
Thus a skip list with n items has height at most $3\log n$ with probability at least $1 - 1/n^2$

L34 : BFS and DFS

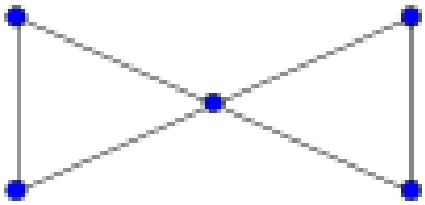
Cut Set
a cut is a partition of the vertices of a graph into two disjoint subsets. The cut-set of the cut is the set of edges whose end points are in different subsets of the partition. Edges are said to be crossing the cut if they are in its cut-set.
A cut is minimum if the size of the cut is not larger than the size of any other cut. Minimum cut is also called the edge connectivity of the Graph.
A cut is maximum if the size of the cut is not smaller than the size of any other cut.



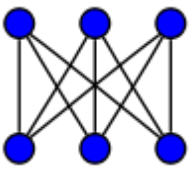


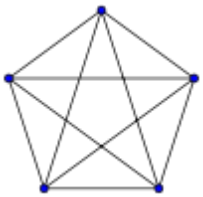
A cut, vertex cut, or separating set of a connected graph G is a set of vertices whose removal renders G disconnected. The connectivity or vertex connectivity κ(G) is the size of a smallest vertex cut. A graph is called k-connected or k-vertex-connected if its vertex connectivity is k or greater.

Euler's formula
If a finite, connected, planar graph is drawn in the plane without any edge intersections, and v is the number of vertices, e is the number of edges and f is the number of faces (regions bounded by edges, including the outer, infinitely-large region), then
v − e + f = 2.

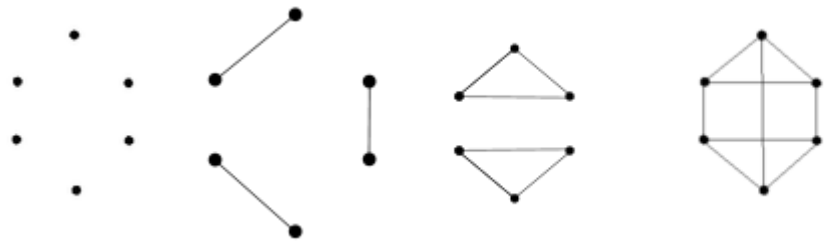


As an illustration, in the butterfly graph given above, v = 5, e = 6 and f = 3.
Kuratowski' K 3,3 K5





Regular graph is a graph without loops and multiple edges where each vertex has the same number of neighbors; i.e. every vertex has the same degree. A regular directed graph must also satisfy the stronger condition that the indegree and outdegree of each vertex are equal to each other. A regular graph with vertices of degree k is called a k-regular graph or regular graph of degree k.

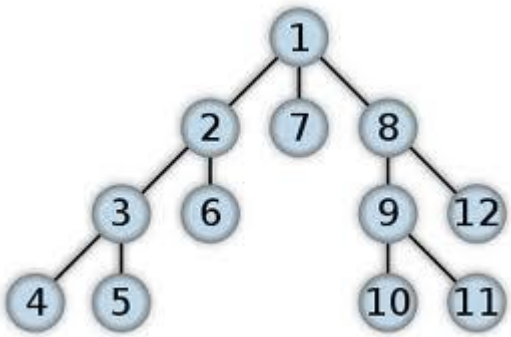


DFS

DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

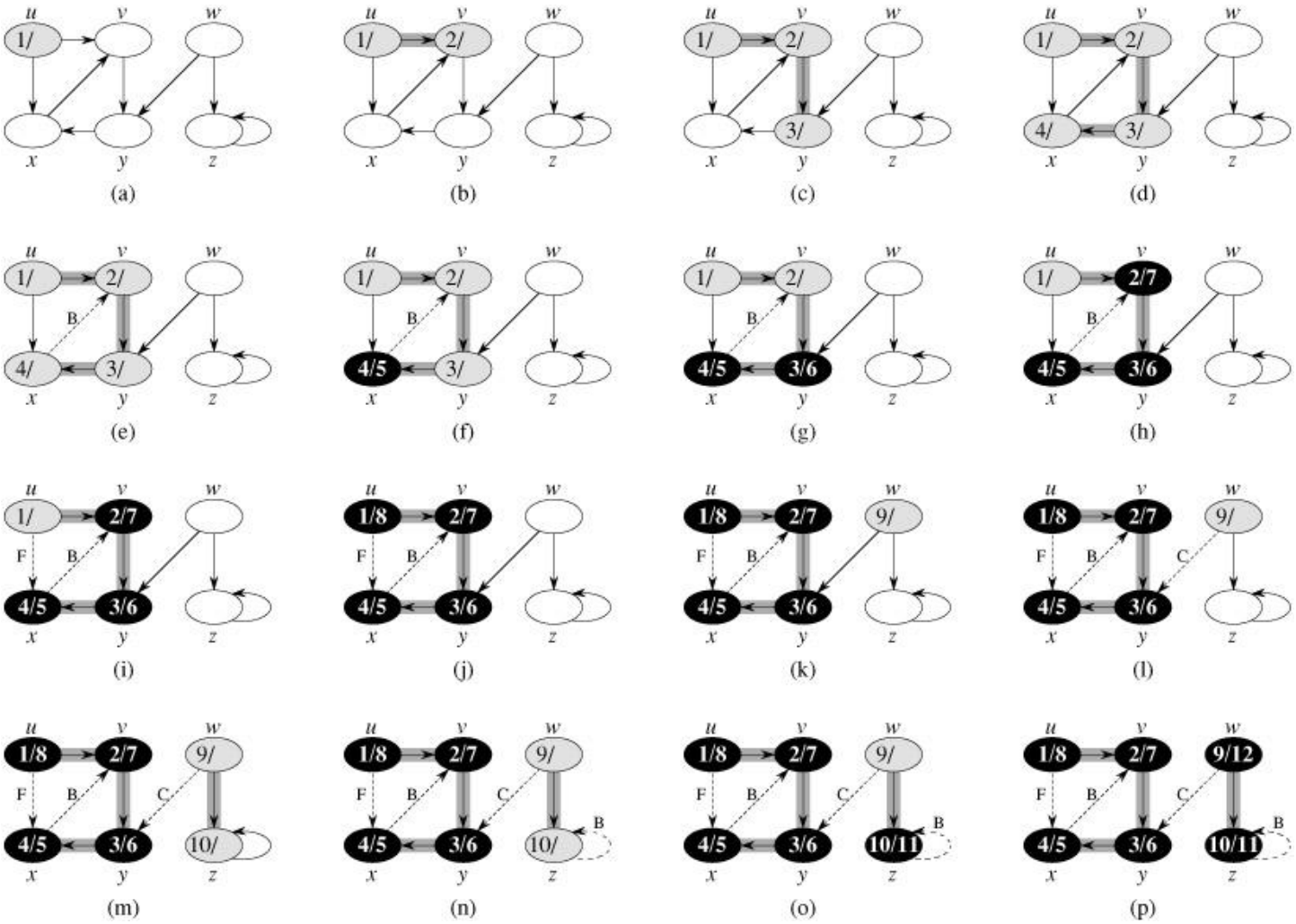
DFS on a graph with ***n*** vertices and ***m*** edges takes ***O***(***n*** + ***m***) time

DFS can be further extended to solve other graph problems.Find and report a path between two given vertices



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

```
dfs(v)
{
dfnumb++;
visited[v] = dfnumb;
for (each edge (v,i) in G)
{
if (visited[i] = =0)
{
parent[i]=v;
dfs(i);
}
else
if (parent[v] != i) && (visited[i] < visited[v])
printf('cycle'%d%d,v,i);
}
```

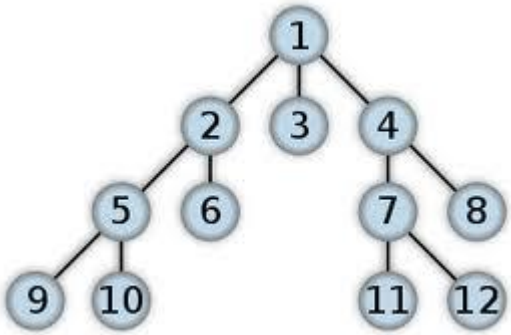


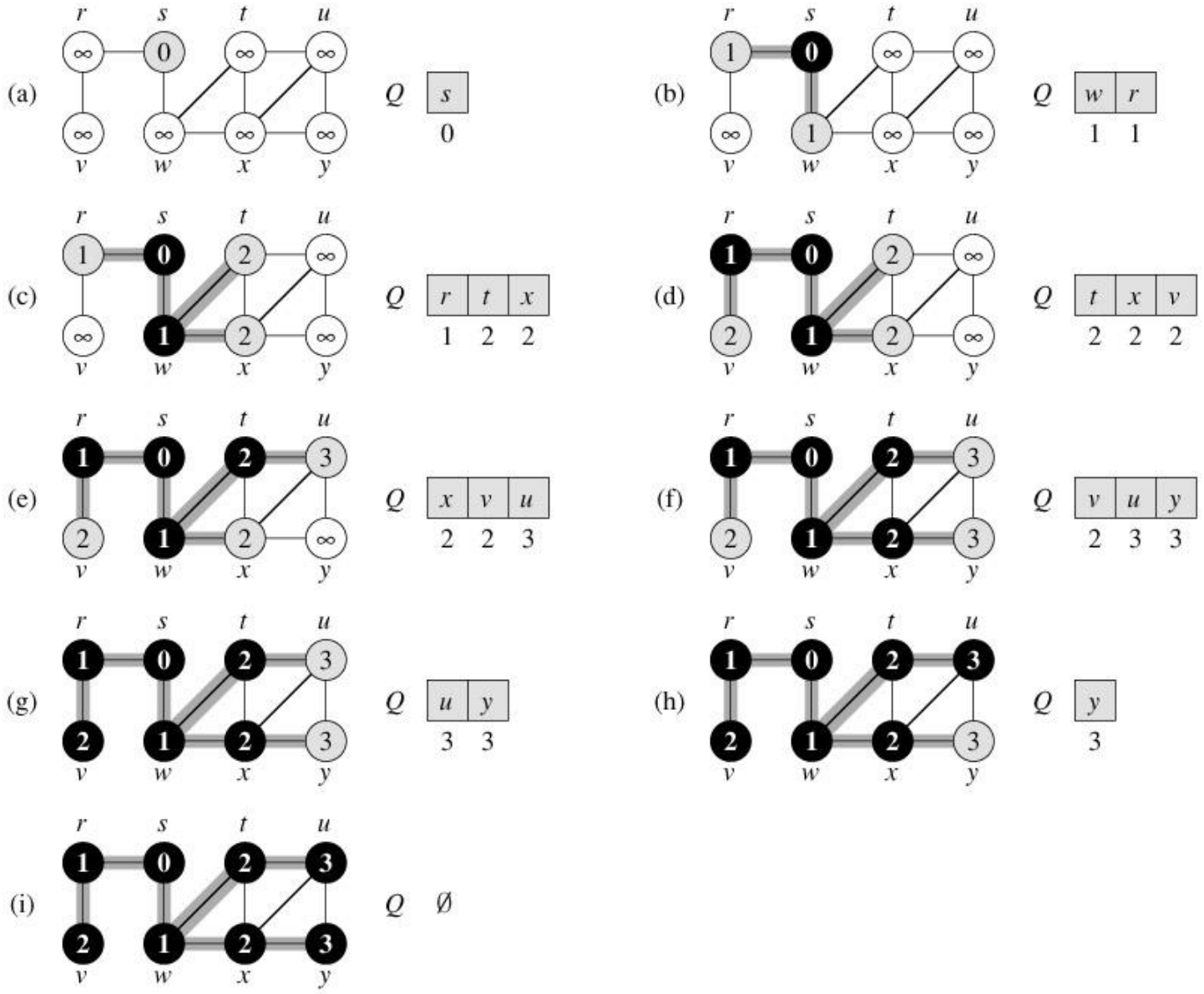
BFS

Given a graph G=(V,E) and a distinguished source vertex s, BFS explores the edges of G to discover every vertex that is reachable from s. It computes the distance from s to each reachable vertex. It also produces a breadth first tree with root s that contains all reachable vertices. The name implies that the algorithm discovers all vertices at distance k from s before discovering any vertices at distance k+1.

BFS on a graph with ***n*** vertices and ***m*** edges takes ***O***(***n*** + ***m***) time

BFS can be further extended to solve other graph problems. Find and report a path with the minimum number of edges between two given vertices. Find a simple cycle, if there is one

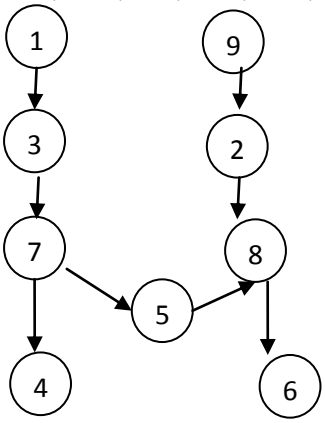




Topological sort

A topological sort or topological ordering of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more topological sorts.

9<2, 3<7,7<5, 5<8, 8<6, 4<6, 1<3, 7<4, 9<5, 2<8 1 9 2 3 7 4 5 8 6



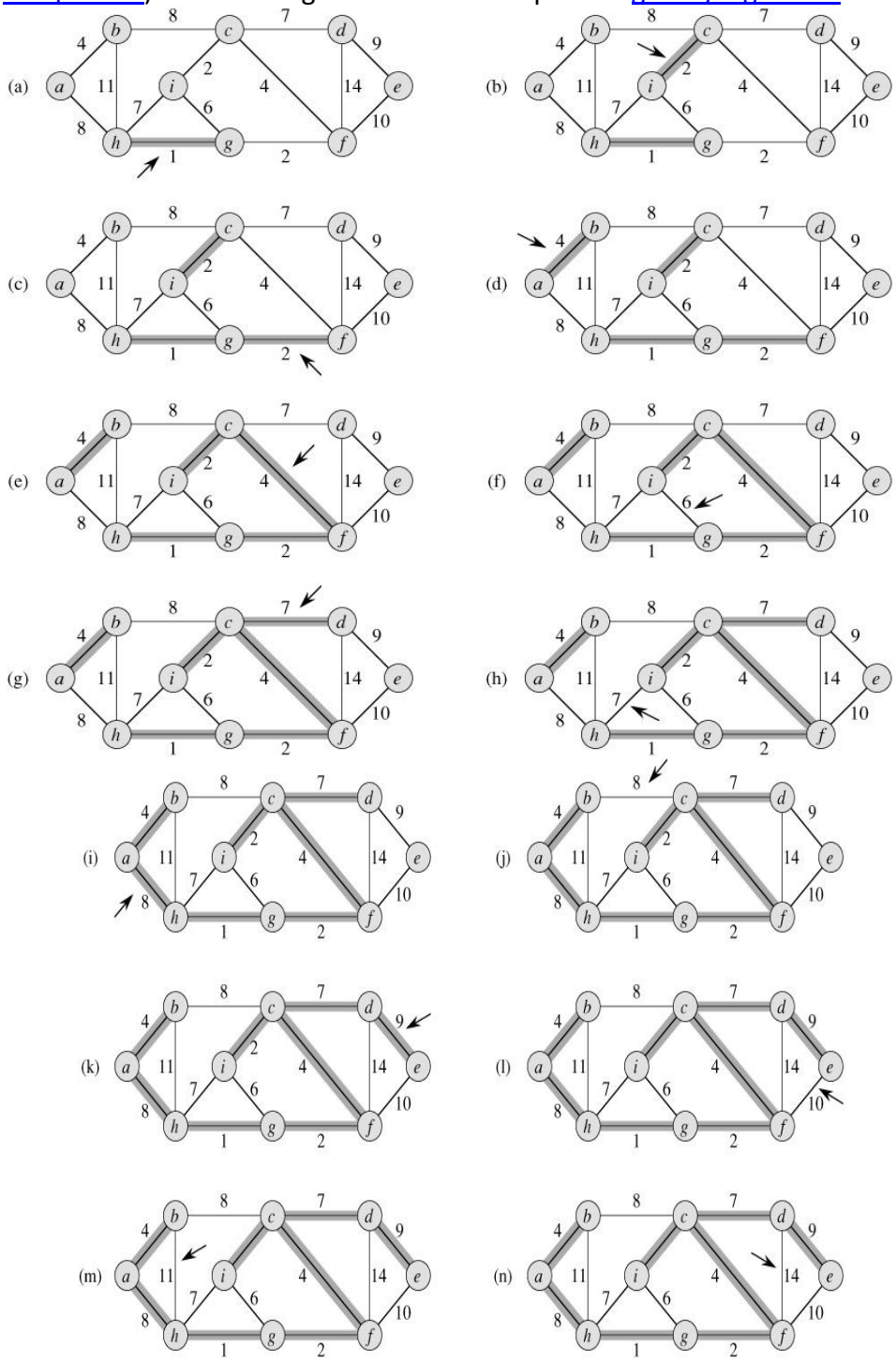
TOPOLOGICAL-SORT(G)
1 call DFS(G) to compute finishing times f[v] for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 return the linked list of vertices

L35: MST Prim Kruskal

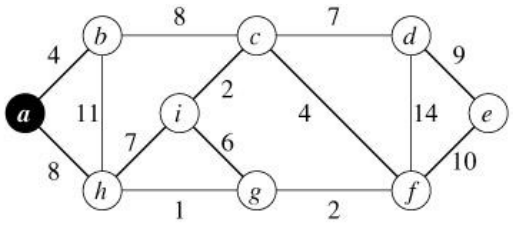
Spanning subgraph
Subgraph of a graph ***G*** containing all the vertices of ***G***
Spanning tree
Spanning subgraph that is itself a (free) tree
Minimum spanning tree (MST)
Spanning tree of a weighted graph with minimum total edge weight

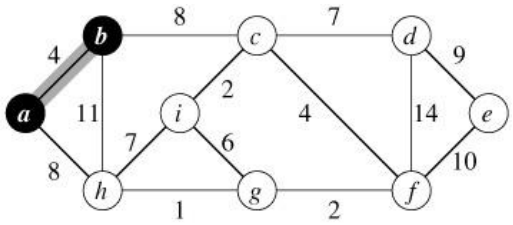
Kruskal

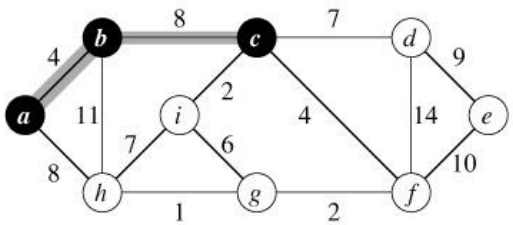
It finds a MST for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm. Complexity: O(E log V)

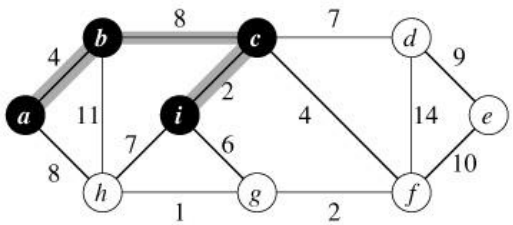


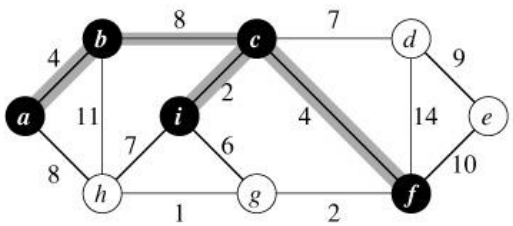
Prim

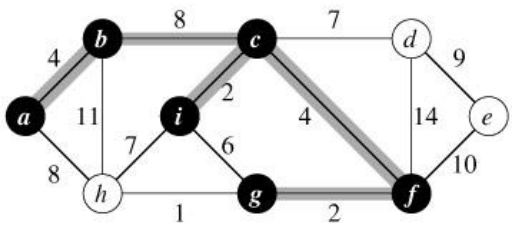
(a)


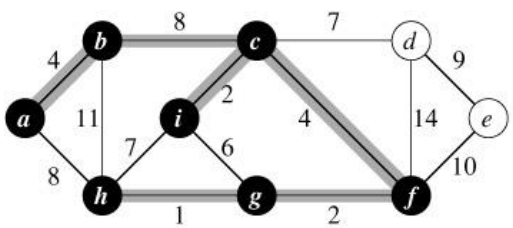
(b)


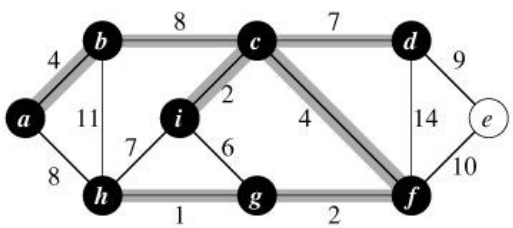
(c)


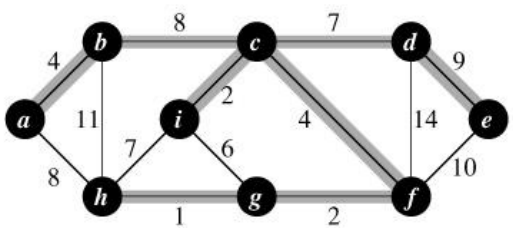
(d)


(e)


(f)


(g)


(h)


(i)


U	Edge(u,v)	V-U
{}		{a,b,c,d,e,f,g,h,i}
{a}	a,b =4 a,h=8	{b,c,d,e,f,g,h,i}
{a,b}	a,h=8 b,c=8 b,h=11	{c,d,e,f,g,h,i}
{a,b,c}	a,h=8 b,h=11 c,d=7 c,f=4 c,i=2	{d,e,f,g,h,i}
{a,b,c,i}	a,h=8 b,h=11 c,d=7 c,f=4 i,g=6	{d,e,f,g,h}

	i,h=7	
{a,b,c,f,,i}	a,h=8 b,h=11 c,d=7 i,g=6 i,h=7 f,d=14 f,e=10 f,g=2	{d,e,g,h}
{a,b,c,f,g,i}	a,h=8 b,h=11 c,d=7 i,g=6(cycle) i,h=7 f,d=14 f,e=10 g,h=1	{d,e,h}
{a,b,c,f,g,h,i}	a,h=8(cycle) b,h=11(cycle) c,d=7 i,g=6(cycle) i,h=7(cycle) f,d=14 f,e=10	{d,e}
{a,b,c,d,f,g,h,i}	a,h=8(cycle) b,h=11(cycle) i,g=6(cycle) i,h=7(cycle) f,d=14(cycle) f,e=10 d,e=9	{e}
{a,b,c,d,f,g,h,i,e}	a,h=8(cycle) b,h=11(cycle) i,g=6(cycle) i,h=7(cycle) f,d=14(cycle) f,e=10(cycle)	{}

Complexity O(ElogV) using min heap

L36: Union-Find DS for Disjoint Sets

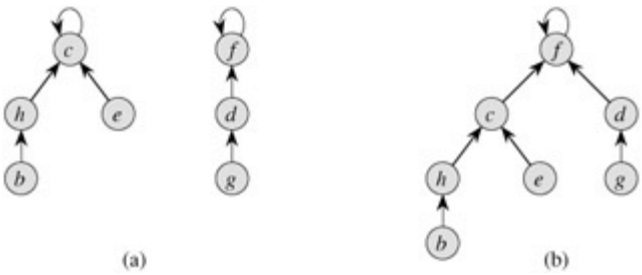
Maintain a dynamic collection of pairwise-disjoint sets S = {S1, S2, …, Sr}.Each set Si has one element distinguished as the representative element, rep[Si].
Must support 3 operations:
• MAKE-SET(x): adds new set {x} to S with rep[{x}] = x (for any x ∉ Si for all i).
• UNION(x, y): replaces sets Sx, Sy with Sx ∪ Sy in S for any x, y in distinct sets Sx, Sy .
• FIND-SET(x): returns representative rep[Sx] of set Sx containing element x.

Analysis
n= total no. of elements, m= total no. of operations
Since Make-Set counts towards total# of operations, m>=n
Can have at most n − 1 UNION operations, since after n − 1 UNIONs, only 1 set remains.
Assume that the first n operations are MAKE-SET (helpful for analysis, usually not really necessary).
Application: dynamic connected components.
For a graph G = (V, E), vertices u, v are in same connected component if and only if there is a path between them.
Connected components partition vertices into equivalence classes.

Store each set Si = {x1, x2, …, xk} as an (unordered) doubly linked list. Define representative element rep[Si] to be the front of the list, x1.
• MAKE-SET(x) initializes x as a lone node. – Θ(1)
• FIND-SET(x) walks left in the list containing x until it reaches the front of the list. – Θ(n)
• UNION(x, y) concatenates the lists containing x and y, leaving rep. as FIND-SET[x]. – Θ(1) if header and trailer nodes are given

Store each set Si = {x1, x2, …, xk} as a balanced tree (ignoring keys). Define representative element rep[Si] to be the root of the tree.
MAKE-SET(x) initializes x as a lone node. – Θ(1)
• FIND-SET(x) walks up the tree containing x until it reaches the root. – Θ(n)
• UNION(x, y) concatenates the trees containing x and y, changing rep. – Θ(n)
Si = {x1, x2, x3, x4, x5}

Two improvements : Union By Rank : Make the root of the smaller tree (fewer nodes) a child of the root of the larger tree.



After this improvement
MAKE-SET(x) initializes x as a lone node. – Θ(1)
• FIND-SET(x) walks up the tree containing x until it reaches the root. – Θ(lgn)
• UNION(x, y) concatenates the trees containing x and y, changing rep. – Θ(lgn)

Path Compression: nodes visited during Find-SET on the trip to the root. Make all nodes on the find path direct children of the root.

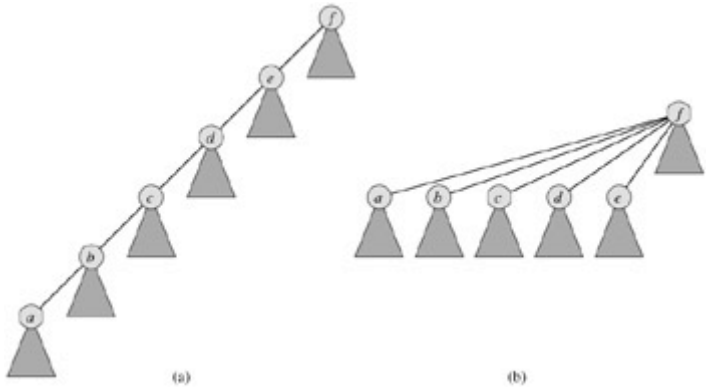
Find-SET makes a pass up to find the root and a pass down as recursion unwinds to update each node on find path to point directly to the root.

After using both union by rank and path compression

O(m α (n))

n	α (n)
0-2	0
3	1
4-7	2
8-2047	3
2048-10^{80}	4

α (n) uses Ackermann function



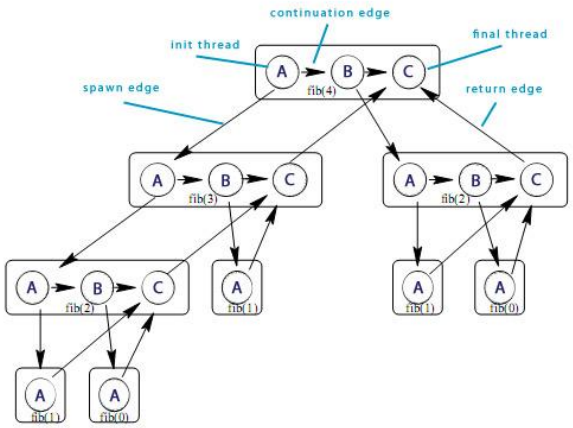
L37: Parallel and Multithreaded algorithms

Dynamic multithreading model
Fibonacci(n):

```
1  if n < 2:                          | thread A
2    return n                         | thread A
3  x = spawn Fibonacci(n-1)           | thread A
4  y = spawn Fibonacci(n-2)           | thread B
5  sync                               | thread C
6  return x + y                       | thread C
```

spawn subroutine can execute at the same time as its parent
sync means wait until all children are done.

This is logical parallelism, not actual. Because a scheduler determines how to map dynamically unfolding execution onto processors.
Parallel instruction stream= Directed acyclic graph (DAG)
The vertices are threads which are maximal sequences of instructions non containing parallel control.
control instructions are spawn, sync, return



So to compute Fib(4), first Fib(3) and Fib(2) have to be computed. Fib(4) spawns one thread to compute Fib(3) and another thread to compute Fib(2) and sync'es. In order to compute Fib(3), values of Fib(2) and Fib(1) have to be known, and to compute Fib(2) values of Fib(1) and Fib(0) have to be known. Fib(3) and Fib(2) spawn their threads and sync. This way the computation unwinds.
T_p - running time of an algorithm on *p* processors,
T_1 - running time of algorithm on 1 processor, and
T_∞ as critical path length, that is the longest time to execute the algorithm on infinite number of processors.

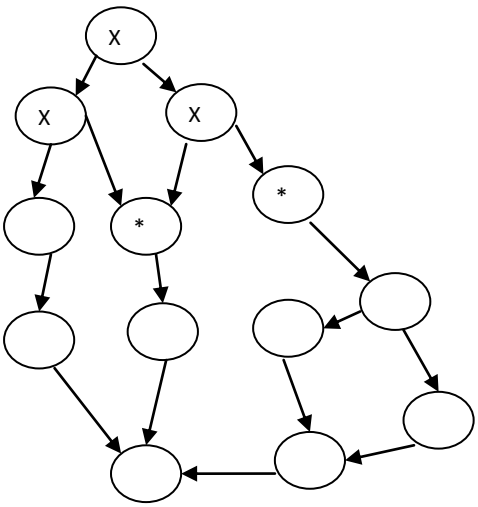
ex: fib(4) has T1=17 (Total threads)
(assuming every thread costing single unit of time)
$T_\infty = 8$ (longest path length)

Based on these measures, the **speedup** of a computation on p processors is T1/Tp, that is how much faster a p processor machine executes the algorithm than a one processor machine. If the speedup is O(p), then we say that the computation has a linear speedup. If, however, speedup > p, then we call it a super-linear speedup. The maximum possible speedup is called **parallelism** and is computed by $T1/T\infty$.

The greedy scheduler schedules as much as it can at every time step. On a p-processor machine every step can be classified into two types. If there are p or more threads ready to execute, the scheduler executes any p of

them (this step is also called a complete step). If there are fewer than p threads, it executes all of them (called incomplete step). The greedy scheduler is an offline scheduler and it's 2-competitive

Proof: Number of complete steps ≤ T1/P
since otherwise more than T1 work would be done.
consider an incomplete step and let G' be a subgraph of G that remains to be executed.
G: P=3



nodes with X are already executed
nodes with * are ready to be executed
Threads with indegree 0 in G' are ready to be executed. Critical path length of G reduces by 1.
The number of incomplete steps is at most T_∞

Matrix multiplication

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$
$$= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

```
Matrix-Add(C, T, n):
  // Adds matrices C and T in-place, producing C = C + T
  // n is power of 2 (for simplicity).
  if  n == 1:
    C[1, 1] = C[1, 1] + T[1, 1]
  else:
    partition C and T into (n/2)x(n/2) submatrices
    spawn Matrix-Add(C11, T11, n/2)
    spawn Matrix-Add(C12, T12, n/2)
    spawn Matrix-Add(C21, T21, n/2)
    spawn Matrix-Add(C22, T22, n/2)
    sync
```

```
Matrix-Multiply(C, A, B, n):
  // Multiplies matrices A and B, storing the result in C.
  // n is power of 2 (for simplicity).
  if  n == 1:
    C[1, 1] = A[1, 1] · B[1, 1]
```

```
  else:
    allocate a temporary matrix T[1...n, 1...n]
    partition A, B, C, and T into (n/2)x(n/2) submatrices
    spawn Matrix-Multiply(C12,A11,B12, n/2)
    spawn Matrix-Multiply(C21,A21,B11, n/2)
    spawn Matrix-Multiply(C22,A21,B12, n/2)
    spawn Matrix-Multiply(T11,A12,B21, n/2)
    spawn Matrix-Multiply(T12,A12,B22, n/2)
    spawn Matrix-Multiply(T21,A22,B21, n/2)
    spawn Matrix-Multiply(T22,A22,B22, n/2)
    sync
    Matrix-Add(C, T, n)
```

Analysis

Let Mp(n) = P processor time for mult

Ap(n)= P processor time for add

Work $A_1(n) = 4\,A_1(n/2) + \theta(1) = O(n^2)$

$M_1(n) = 8\,M_1(n/2) + \theta(n^2) = \theta(n^3)$

Critical Path length

$A_\infty(n) = A_\infty(n/2) + \theta(1) = \theta(\lg n)$

$M_\infty(n) = M_\infty(n/2) + \theta(\lg n) = \theta(\lg^2 n)$

Parallelism = $M_1(n)/M_\infty(n) = \theta(n^3)/(\lg^2 n)$

For 1000X1000 matrices

Parallelism ≈ $1000^3/10^2 = 10^7$ parallelism //blue jean has more than 10000 processors

parallelism is much bigger than the typical p so We can trade parallelism for space efficiency

```
Matrix-MultAdd(C, A, B, n):
  // Multiplies matrices A and B, storing the result in C.
  // n is power of 2 (for simplicity).
  if n == 1:
    C[1, 1] = A[1, 1] · B[1, 1]
  else:
    partition A, B, C into (n/2)x(n/2) submatrices
    spawn Matrix-Multadd(C12,A11,B12, n/2)
    spawn Matrix-Multadd(C21,A21,B11, n/2)
    spawn Matrix-Multadd(C22,A21,B12, n/2)
sync
    spawn Matrix-Multadd(C11,A12,B21, n/2)
    spawn Matrix-Multadd(C12,A12,B22, n/2)
    spawn Matrix-Multadd(C21,A22,B21, n/2)
    spawn Matrix-Multadd(C22,A22,B22, n/2)
    sync
```

Critical Path length

$MA_\infty(n) = 2MA_\infty(n/2) + \theta(1) = O(n)$

Parallelism = $M_1(n)/M_\infty(n) = O(n^3)/O(n) = O(n^2)$

For 1000X1000 matrices Parallelism ≈ $1000^3/1000 = 10^6$ parallelism This is faster and takes less space

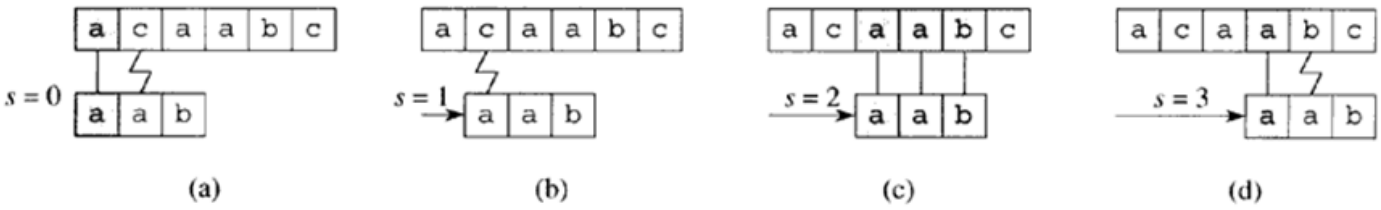
L38: String Matching Algorithms

A string is a sequence of characters
Examples of strings: C++ program, HTML document, DNA sequence, Digitized image
An alphabet S is the set of possible characters for a family of strings
Example of alphabets:
ASCII (used by C and C++), Unicode (used by Java), {0, 1}, {A, C, G, T}
Let P be a string of size m
A substring P[i .. j] of P is the subsequence of P consisting of the characters with ranks between i and j
A prefix of P is a substring of the type P[0 .. i]
A suffix of P is a substring of the type P[i ..m - 1]
Given strings T (text) and P (pattern), pattern matching problem consists of finding a substring of T equal to P
Applications: Text editors, Search engines, Biological research

Naïve Algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1..m] = T[s+1..s+m]$ for each of the $n - m + 1$ possible values of s.

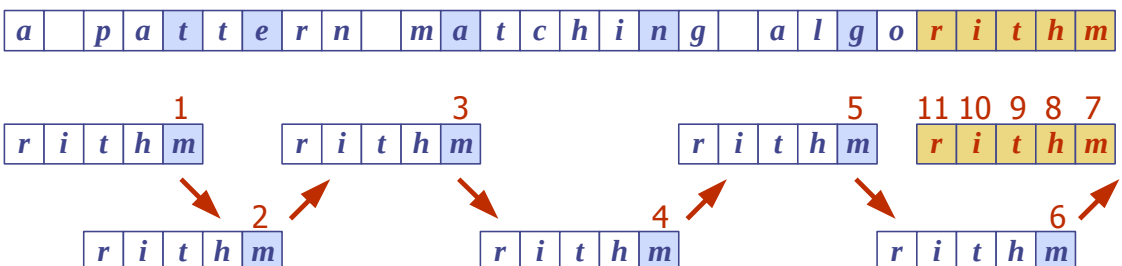
```
NAIVE-STRING-MATCHER(T, P)
1  n ← length[T]
2  m ← length[P]
3  for s ← 0 to n − m
4       do if P[1..m] = T[s + 1..s + m]
5             then print "Pattern occurs with shift" s
```



Worst case
T = aaa … ah
P = aaah

Boyer-Moore Algorithm

Boyer-Moore's pattern matching algorithm is based on two heuristics
Looking-glass heuristic: Compare P with a subsequence of T moving backwards
Character-jump heuristic: When a mismatch occurs at T[i] = c
If P contains c, shift P to align the last occurrence of c in P with T[i]
Else, shift P to align P[0] with T[i + 1]



Boyer-Moore's algorithm runs in time ***O***(***nm*** + ***s***)
Example of worst case:
T = ***aaa … a***
P = ***baaa***
The worst case may occur in images and DNA sequences but is unlikely in English text
Boyer-Moore's algorithm is significantly faster than the brute-force algorithm on English text

Rabin-Karp Algorithm

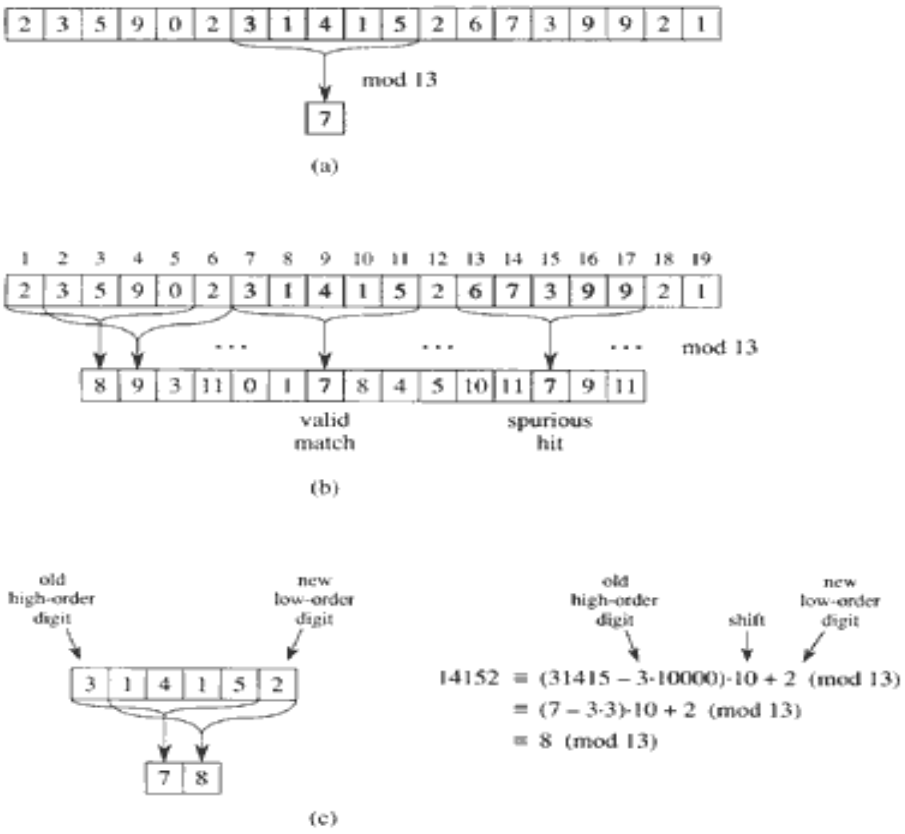


Figure 32.5 The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13. **(a)** A text string. A window of length 5 is shown shaded. The numerical value of the shaded number is computed modulo 13, yielding the value 7. **(b)** The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern $P = 31415$, we look for windows whose value modulo 13 is 7, since $31415 \equiv 7 \pmod{13}$. Two such windows are found, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit. **(c)** Computing the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 gives us the new value 14152. All computations are performed modulo 13, however, so the value for the first window is 7, and the value computed for the new window is 8.

L39: Knuth Morris Pratt Algorithm

It never re-compares a text symbol that has matched a pattern symbol. As a result, complexity of the searching phase of the KMP = $O(n)$. Preprocessing phase has a complexity of $O(m)$. Since $m \leq n$, the overall complexity of is $O(n)$. A border of x is a substring that is both proper prefix and proper suffix of x. We call its length b the width of the border.

Let x = abacab. The proper prefixes of x are ε, a, ab, aba, abac, abaca

The proper suffixes of x are ε, b, ab, cab, acab, bacab The borders of x are ε, ab

The border ε has width 0, the border ab has width 2

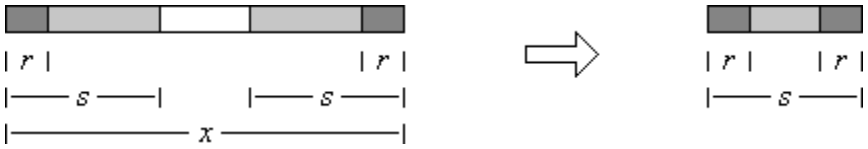
```
0 1 2 3 4 5 6 7 8 9 ...
a b c a b c a b d
a b c a b d
      a b c a b d
```

The symbols at positions 0, ..., 4 have matched. Comparison c-d at position 5 yields a mismatch. The pattern can be shifted by 3 positions, and comparisons are resumed at position 5. The shift distance is determined by the widest border of the matching prefix of p. In this example, the matching prefix is abcab, its length is $j = 5$. Its widest border is ab of width $b = 2$. The shift distance is $j - b = 5 - 2 = 3$.

In the preprocessing phase, the width of the widest border of each prefix of the pattern is determined. Then in the search phase, the shift distance can be computed according to the prefix that has matched

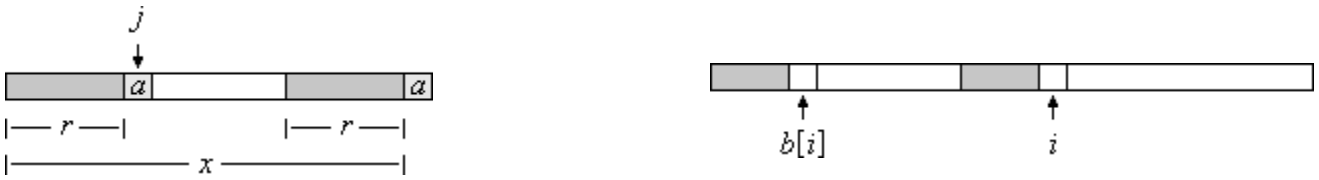
Theorem: Let r, s be borders of a string x, where |r| < |s|. Then r is a border of s.

Proof: Since r is a prefix of x, it is also a proper prefix of s, because it is shorter than s. But r is also a suffix of x and, therefore, proper suffix of s. Thus r is a border of s.



if s is the widest border of x, the next-widest border r of x is obtained as the widest border of s etc.

Definition: Let x be a string and $a \in A$ a symbol. A border r of x can be extended by a, if ra is a border of xa.



border r of width j of x can be extended by a, if $x_j = a$.

In preprocessing phase an array b of length $m+1$ is computed. Each entry $b[i]$ contains width of widest border of the prefix of length i of pattern ($i = 0, ..., m$). Since the prefix ε of length $i = 0$ has no border, we set $b[0] = -1$. Provided that the values $b[0], ..., b[i]$ are already known, the value of $b[i+1]$ is computed by checking if a border of the prefix $p_0 ... p_{i-1}$can be extended by symbol p_i. This is case if $p_{b[i]} = p_i$ The borders to be examined are obtained in decreasing order from values $b[i]$, $b[b[i]]$ etc.

The preprocessing algorithm comprises a loop with a variable j assuming these values. A border of width j can be extended by p_i, if$p_j = p_i$. If not, the next-widest border is examined by setting $j = b[j]$. The loop terminates at the latest if no border can be extended ($j = -1$).

After increasing j by the statement `j++` in each case j is the width of the widest border of $p_0 ... p_i$. This value is written to $b[i+1]$ (to $b[i]$ after increasing i by the statement `i++`).

Preprocessing Algorithm

```
 void kmpPreprocess()
{
   int i=0, j=-1;
   b[i]=j;
```

```
   while (i<m)
   {
      while (j>=0 && p[i]!=p[j]) j=b[j];
      i++; j++;
      b[i]=j;
   }
}
```

For pattern *p* = ababaa the widths of the borders in array *b* have the following values. For instance we have *b*[5] = 3, since the prefix ababa of length 5 has a border of width 3.

j:	0	1	2	3	4	5	6
p[*j*]:	a	b	a	b	a	a	
b[*j*]:	-1	0	0	1	2	3	1

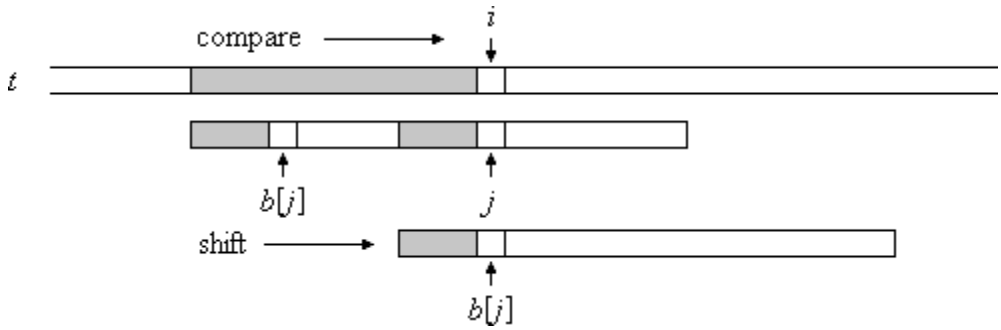
Searching algorithm

Conceptually, the above preprocessing algorithm could be applied to the string *pt* instead of *p*. If borders up to a width of *m* are computed only, then a border of width *m* of some prefix *x* of *pt* corresponds to a match of the pattern in *t* (provided that the border is not self-overlapping)

This explains the similarity between the preprocessing algorithm and the following searching algorithm.

When in the inner while loop a mismatch at position *j* occurs, the widest border of the matching prefix of length *j* of the pattern is considered. Resuming comparisons at position *b*[*j*], the width of the border, yields a shift of the pattern such that the border matches. If again a mismatch occurs, the next-widest border is considered, and so on, until there is no border left (*j* = -1) or the next symbol matches. Then we have a new matching prefix of the pattern and continue with the outer while loop.

```
void kmpSearch()
{
   int i=0, j=0;
   while (i<n)
   {
      while (j>=0 && t[i]!=p[j]) j=b[j];
      i++; j++;
      if (j==m)
      {
         report(i-j);
         j=b[j];
      }
   }
}
```



If all *m* symbols of the pattern have matched the corresponding text window (*j* = *m*), a function *report* is called for reporting the match at position *i-j*. Afterwards, the pattern is shifted as far as its widest border allows.

In example where matches are drawn in green and mismatches in red.

```
0 1 2 3 4 5 6 7 8 9 ...
a b a b b a b a a
a b a b a c
    a b a b a c
        a b a b a c
          a b a b a c
              a b a b a c
```

Analysis : The inner while loop of the preprocessing algorithm decreases the value of *j* by at least 1, since *b*[*j*] < *j*. The loop terminates at the latest when *j* = -1, therefore it can decrease the value of *j* at most as often as it has been increased previously by `j++`. Since `j++` is executed in the outer loop exactly *m* times, the overall number of executions of the inner while loop is limited to *m*. The preprocessing algorithm therefore requires *O*(*m*) steps. From similar arguments it follows that the searching algorithm requires *O*(*n*) steps. The above example illustrates this: the comparisons (green and red symbols) form "stairs". The whole staircase is

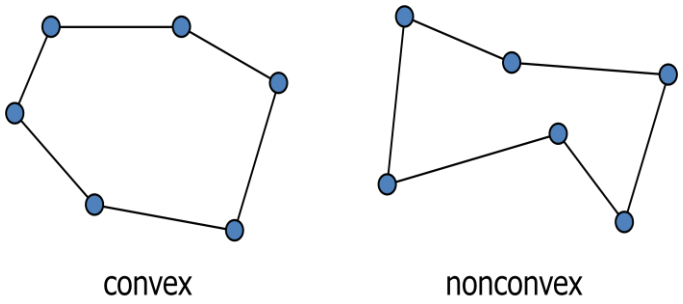
at most as wide as it is high, so at most 2*n*comparisons are performed. Since $m \leq n$ the overall complexity of the KMP is in $O(n)$.

Backtracking is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution.
Branch and bound (BB) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded en masse, by using upper and lower estimated bounds of the quantity being

L40: Convex Hull & Closest Pair Problem

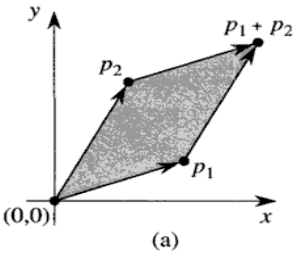
A convex polygon is a nonintersecting polygon whose internal angles are all convex (i.e., less than p)
In a convex polygon, a segment joining two vertices of the polygon lies entirely inside the polygon

The convex hull of a set of points is the smallest convex polygon containing the points
Think of a rubber band snapping around the points
The convex hull is a segment
Two points : All the points are collinear
The convex hull is a point
there is one point : All the points are coincident
Motion planning
Find an optimal route that avoids obstacles for a robot
Geometric algorithms
Convex hull is like a two-dimensional sorting



The following method computes the convex hull of a set of points
Phase 1: Find the lowest point (anchor point)
Phase 2: Form a nonintersecting polygon by sorting the points counterclockwise around the anchor point
Phase 3: While the polygon has a nonconvex vertex, remove it
The orientation of three points in the plane is clockwise, counterclockwise, or collinear
orientation(a, b, c)
clockwise (CW, right turn)
counterclockwise (CCW, left turn)
collinear (COLL, no turn)
The orientation of three points is characterized by the sign of the determinant D(a, b, c), whose absolute value is twice the area of the triangle with vertices a, b and c

$$\Delta(a,b,c) = \begin{vmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{vmatrix}$$



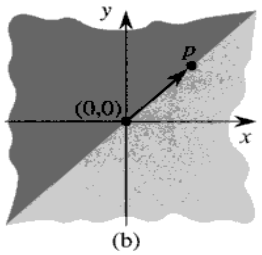
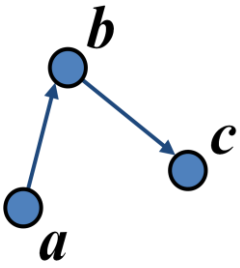
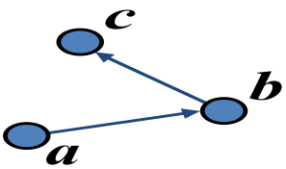


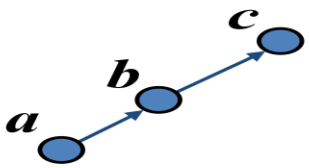
Figure 33.1 **(a)** The cross product of vectors p_1 and p_2 is the signed area of the parallelogram. **(b)** The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region contains vectors that are counterclockwise from p.



CW



CCW



Testing whether a vertex is convex can be done using the orientation function
Let p, q and r be three consecutive vertices of a polygon, in counterclockwise order
q convex ⇔ orientation(p, q, r) = CCW
q nonconvex ⇔ orientation(p, q, r) = CW or COLL

If $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ then

$$p_1 \times p_2 = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = x_1y_2 - x_2y_1$$ *

If $p_1 \times p_2 > 0$, then p1 is clockwise from p2 (turning around the origin);
if $p_1 \times p_2 < 0$, then p1 is counterclockwise from p2;
If $p_1 \times p_2 = 0$, then p1 and p2 are collinear (pointing in either same or opposite directions)
Computing the convex hull of a set of points takes $O(n \log n)$ time
Finding the anchor point takes $O(n)$ time
Sorting the points counterclockwise around the anchor point takes $O(n \log n)$ time
Use the orientation comparator and any sorting algorithm that runs in $O(n \log n)$ time (e.g., heap-sort or merge-sort)
The Graham scan takes $O(n)$ time
Each point is inserted once in sequence ***H***
Each vertex is removed at most once from sequence ***H***

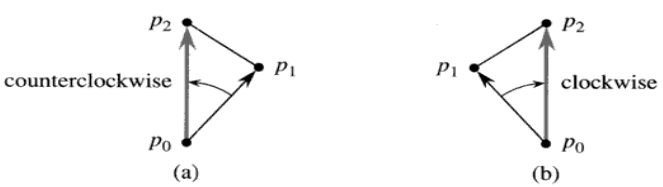
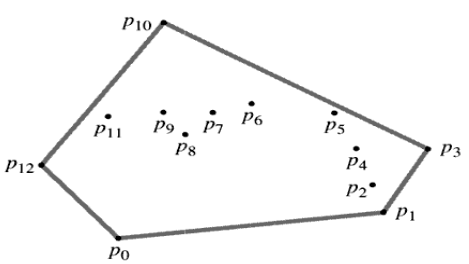


Figure 33.2 Using the cross product to determine how consecutive line segments $\overline{p_0p_1}$ and $\overline{p_1p_2}$ turn at point p_1. We check whether the directed segment $\overrightarrow{p_0p_2}$ is clockwise or counterclockwise relative to the directed segment $\overrightarrow{p_0p_1}$. **(a)** If counterclockwise, the points make a left turn. **(b)** If clockwise, they make a right turn.

DIRECTION(p_i, p_j, p_k)
1 **return** $(p_k - p_i) \times (p_j - p_i)$



```
GRAHAM-SCAN(Q)
 1  let p_0 be the point in Q with the minimum y-coordinate,
          or the leftmost such point in case of a tie
 2  let ⟨p_1, p_2, ..., p_m⟩ be the remaining points in Q,
          sorted by polar angle in counterclockwise order around p_0
          (if more than one point has the same angle, remove all but
          the one that is farthest from p_0)
 3  PUSH(p_0, S)
 4  PUSH(p_1, S)
 5  PUSH(p_2, S)
 6  for i ← 3 to m
 7      do while the angle formed by points NEXT-TO-TOP(S), TOP(S),
                  and p_i makes a nonleft turn
 8             do POP(S)
 9         PUSH(p_i, S)
10  return S
```

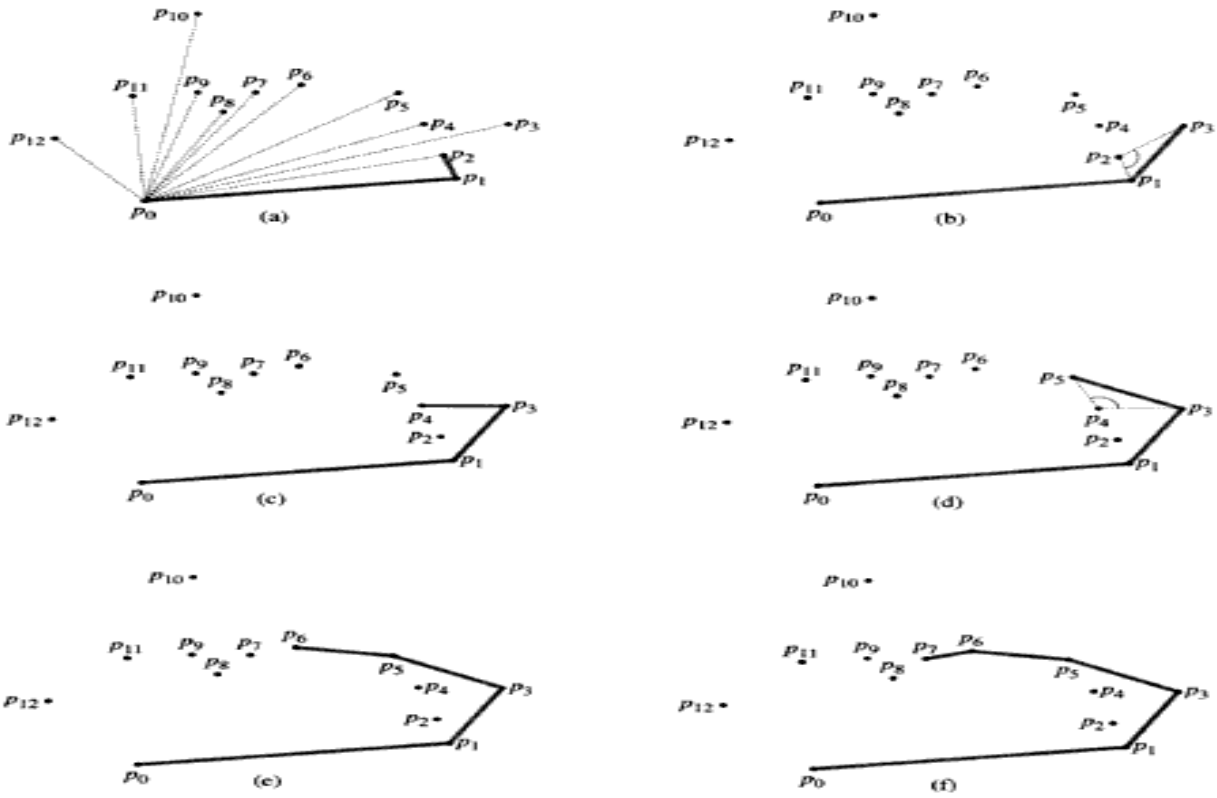


Figure 33.7 The execution of GRAHAM-SCAN on the set Q of Figure 33.6. The current convex hull contained in stack S is shown in gray at each step. **(a)** The sequence $\langle p_1, p_2, \ldots, p_{12} \rangle$ of points numbered in order of increasing polar angle relative to p_0, and the initial stack S containing p_0, p_1, and p_2. **(b)–(k)** Stack S after each iteration of the **for** loop of lines 6–9. Dashed lines show nonleft turns, which cause points to be popped from the stack. In part (h), for example, the right turn at angle $\angle p_7 p_8 p_9$ causes p_8 to be popped, and then the right turn at angle $\angle p_6 p_7 p_9$ causes p_7 to be popped. **(l)** The convex hull returned by the procedure, which matches that of Figure 33.6.