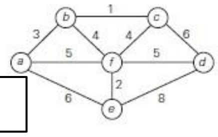
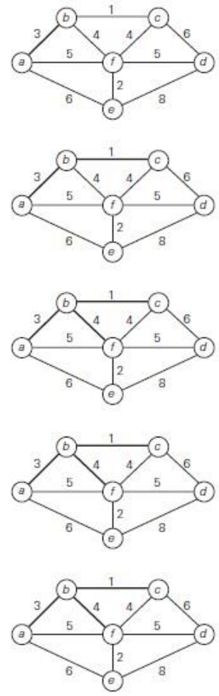
Prim Algorithm

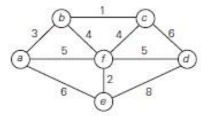


Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

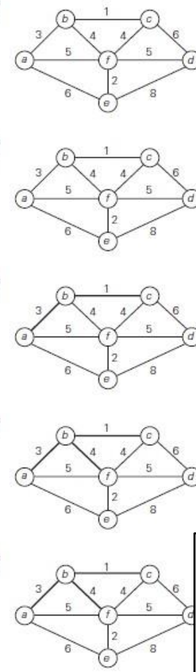


Cycle checking: a cycle is created if added edge connects vertices in the same connected component

Kruskal



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		



{{bc}}
{{bc},{ef}}
{{bc,ab},{ef}}
{{bc,ab,ef,bf}}
{{bc,ab,ef,bf,df}}

The list of keys: 30, 20, 56, 75, 31, 19
The hash function: $h(K) = K \bmod 11$
The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

0	1	2	3	4	5	6	7	8	9	10
								30		
								30	20	
	56							30	20	
	56							30	20	75
31	56							30	20	75
31	56	19						30	20	75

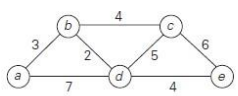
b. The largest number of key comparisons in a successful search is 6 (when searching for $K = 19$).

Searching in Binary Search Tree

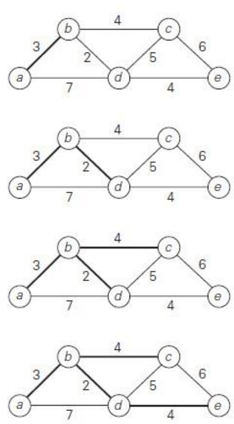
Algorithm ***BTS(x, v)***
//Searches for node with key equal to v in BST rooted at node x
if x = NIL return -1
else if v = K(x) return x
else if v < K(x) return ***BTS (left(x), v)***
else return ***BTS (right(x), v)***

Efficiency:
worst case: $C(n) = n$
average case: $C(n) \approx 2\ln n \approx 1.39 \log_2 n$

Dijkstra



Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		



ALGORITHM *Height(T)*
//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \varnothing$ **return** -1
else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$

$A(n(T)) = A(n(T_{left})) + A(n(T_{right})) + 1$ for $n(T) > 0$,
$A(0) = 0$.

So, after making $n + 1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will be left with the strictly increasing array $A[1..n-1]$ to sort. This sorting of strictly increasing arrays of diminishing sizes will continue until the last one $A[n-2..n-1]$ has been processed. The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

$$C_{avg}(n) \approx 2n \ln n \approx 1.39 n \log_2 n.$$

Quicksort

Algorithm *DivCongPower(a, n)*
//Computes a^n by a divide-and-conquer algorithm
//Input: A number a and a positive integer n
//Output: The value of a^n
if $n = 1$ **return** a
else return $DivCongPower(a, \lfloor n/2 \rfloor) * DivCongPower(a, \lceil n/2 \rceil)$

3. The algorithm fills a table with $n+1$ rows and $W+1$ columns, spending $\Theta(1)$ time to fill one cell (either by applying (8.6) or (8.7)). Hence, its time efficiency and its space efficiency are in $\Theta(nW)$.

In order to identify the composition of an optimal subset, the algorithm repeatedly compares values at no more than two cells in a previous row. Hence, its time efficiency class is in $O(n)$.

3. For the bottom-up dynamic programming algorithm for the knapsack problem, prove that
a. its time efficiency is ?(nW).
b. its space efficiency is ?(nW).
c. the time needed to find the composition of an optimal subset from a filled dynamic programming table is O(n).

Binary Search

Very efficient algorithm for searching in sorted array:

K vs $A[0] \ldots A[m] \ldots A[n-1]$

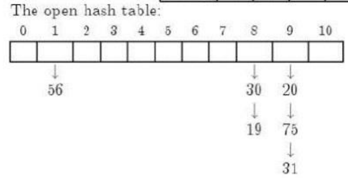
If $K = A[m]$, stop (successful search);
otherwise, continue searching by the same method in $A[0..m-1]$ if $K < A[m]$ and in $A[m+1..n-1]$ if $K > A[m]$

ALGORITHM *BinarySearch*($A[0..n-1]$, K)
//Implements nonrecursive binary search
//Input: An array $A[0..n-1]$ sorted in ascending order and
// a search key K
//Output: An index of the array's element that is equal to K
// or -1 if there is no such element
$l \leftarrow 0$; $r \leftarrow n - 1$
while $l \le r$ **do**
$m \leftarrow \lfloor (l + r)/2 \rfloor$
if $K = A[m]$ **return** m
else if $K < A[m]$ $r \leftarrow m - 1$
else $l \leftarrow m + 1$
return -1

1. a.
The list of keys: 30, 20, 56, 75, 31, 19
The hash function: $h(K) = K \bmod 11$
The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

The open hash table:



b. The largest number of key comparisons in a successful search in this table is 3 (in searching for $K = 31$).

c. The average number of key comparisons in a successful search in this table, assuming that a search for each of the six keys is equally likely, is

$$\frac{1}{6}\cdot 1 + \frac{1}{6}\cdot 1 + \frac{1}{6}\cdot 1 + \frac{1}{6}\cdot 2 + \frac{1}{6}\cdot 3 + \frac{1}{6}\cdot 2 = \frac{10}{6} \approx 1.7.$$

A - 1 B - 2 C - 3 D - 4 E - 5 F - 6 G - 7 H - 8
I - 9 J - 10 K - 11 L - 12 M - 13 N - 14
O - 15
P - 16
Q - 17
R - 18
S - 19
T - 20
U - 21
V - 22
W - 23
X - 24
Y - 25
Z - 26

Space-for-time tradeoffs

Two varieties of space-for-time algorithms:
• input enhancement —preprocess the input (or its part) to store some info to be used later in solving the problem • counting sorts • string searching algorithms • prestructuring—preprocess the input to make accessing its elements easier • hashing • indexing schemes (e.g., B-trees)

Open hashing • each cell is a header of linked list of all keys hashed to it
Closed hashing • one key per cell • in case of collision, finds another cell by: • linear probing: use next free bucket • double hashing: use second hash function to compute increment

Comparison Sorting

Array $A[0..5]$		62	31	84	96	19	47
Initially	Count []	0	0	0	0	0	0
After pass $i = 0$	Count []	3	0	1	1	0	0
After pass $i = 1$	Count []		1	2	2	0	1
After pass $i = 2$	Count []			4	3	0	1
After pass $i = 3$	Count []				5	0	1
After pass $i = 4$	Count []					0	2
Final state	Count []	3	1	4	5	0	2
Array $S[0..5]$		19	31	47	62	84	96

Hoorspool

character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R             B A R B E R
        B A R B E R          B A R B E R
          B A R B E R              B A R B E R
```

A B O X
1 2 3 6

```
text:    BARD LOVED BANANAS
pattern:BAOBAB
             BAOBAB
               BAOBAB
                   BAOBAB (unsuccessful search)
```

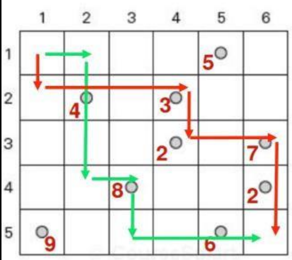
DP solution to the coin-row problem (cont.)

$F(n) = max\{c_n + F(n\text{-}2),\ F(n\text{-}1)\}$ for $n > 1$,
$F(0) = 0,\ F(1)=c_1$

• `dp[2] = max(dp[1], coin[2] + dp[0]) = max(5, 1 + 0) = 5`
• `dp[3] = max(dp[2], coin[3] + dp[1]) = max(5, 2 + 5) = 7`
• `dp[4] = max(dp[3], coin[4] + dp[2]) = max(7, 10 + 5) = 15`
• `dp[5] = max(dp[4], coin[5] + dp[3]) = max(15, 6 + 7) = 15`
• `dp[6] = max(dp[5], coin[6] + dp[4]) = max(15, 2 + 15) = 17`

```
ShiftTable(P[0..m − 1])        //generate Table of shift
i ← m − 1                      //position of the pattern
while i ≤ n − 1 do
    k ← 0                      //number of matched ch
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```

Coin-Collecting Problem: Ex-1



	1	2	3	4	5	6
1	0	0	0	0	5	5
2	0	4	4	7	7	7
3	0	4	4	9	9	16
4	0	4	12	12	12	18
5	9	9	12	12	18	18

$F(i,j) = max\{F(i\text{-}1, j),\ F(i, j\text{-}1)\} + c_{ij}$ for $1 \le i \le n,\ 1 \le j \le m$
where $c_{ij} = 1$ if there is a coin in cell (i,j), and $c_{ij} = 0$ otherwise
$F(0, j) = 0$ for $1 \le j \le m$ and $F(i, 0) = 0$ for $1 \le i \le n$.

	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	**5**

RobotCollectingRobot

```
ALGORITHM RobotCoinCollection(C[1..n, 1..m])
//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an n × m board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix C[1..n, 1..m] whose elements are equal to 1 and 0
//for cells with and without a coin, respectively
//Output: Largest number of coins the robot can bring to cell (n, m)
F[1, 1] ← C[1, 1];   for j ← 2 to m do F[1, j] ← F[1, j − 1] + C[1, j]
for i ← 2 to n do
    F[i, 1] ← F[i − 1, 1] + C[i, 1]
    for j ← 2 to m do
        F[i, j] ← max(F[i − 1, j], F[i, j − 1]) + C[i, j]
return F[n, m]
```

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

Closed Hashing

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

Knapsack of capacity $W = 5$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

	i	capacity j 0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	**37**

$$F(i, j) = \begin{cases} \max\{F(i-1, j),\ v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \ge 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

It is convenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \ge 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \ge 0.$$

The maximal value of a feasible subset is $F[5, 6] = 65$. The optimal subset is {item 3, item 5}.

Using the data in the first six columns, we conclude that the largest amount of money that can be obtained for the input 5, 1, 2, 10, 6 is $F(5) = 15$, which is obtained by taking coins $c_4 = 10$ and $c_1 = 5$.

$F(n) = max\{c_n + F(n\text{-}2),\ F(n\text{-}1)\}$ for $n > 1$,
$F(0) = 0,\ F(1)=c_1$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	17

index	0	1	2	3	4	5	6
coins	--	5	1	2	10	6	2
F()	0	5	5	7	15	15	17

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \ge 0$ in recurrence (5.1), then

$$T(n) = aT(n/b) + f(n),$$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

$$C_{worst}(n) = \sum_{i=1}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

4. a. The worst case: e.g., searching for the pattern $\underbrace{10\ldots0}_{m-1}$ in the text of n 0's. $C_w = m(n - m + 1)$.

b. The best case: e.g., searching for the pattern $\underbrace{0\ldots0}_{m}$ in the text of n 0's. $C_b = m$.

5. Yes: e.g., for the pattern $\underbrace{10\ldots0}_{m-1}$ and the text $\underbrace{0\ldots0}_{n}$, $C_{bf} = n - m + 1$ while $C_{Horspool} = m(n - m + 1)$.

b. For the pattern 10000, the shift table is

c	0	1
$t(c)$	1	4

The algorithm will make four successful and one unsuccessful comparison and then shift the pattern one position to the right on each of its trials:

```
0 0 0 0 0 0                          0 0 0 0 0
1 0 0 0 0
  1 0 0 0 0
              etc.
                                     1 0 0 0 0
```

The total number of character comparisons will be $C = 5 \cdot 996 = 4980$.

```
ALGORITHM InsertionSort(A[0..n − 1])
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

$$A(2^0) = 0.$$

Now backward substitutions encounter no problems:

$A(2^k) = A(2^{k-1}) + 1$ substitute $A(2^{k-1}) = A(2^{k-2}) + 1$
$= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2$ substitute $A(2^{k-2}) = A(2^{k-3}) + 1$
$= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3$ ...
...
$= A(2^{k-i}) + i$
...
$= A(2^{k-k}) + k.$

Backward substitution

Thus, we end up with

$$A(2^k) = A(1) + k = k,$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

In fact, one can prove (Problem 7 in this section's exercises) that the exact solution for an arbitrary value of n is given by just a slightly more refined formula $A(n) = \lfloor \log_2 n \rfloor$.

item 4 is included in the optimal solution since the value goes up from 32 to 37. Find the items in $F(4\text{-}1, 5\text{-}2) = F(3,3)$