Summary Algorithms And Complexity: Lecture(s) all

GROWTH RATE, COMPLEXITY

log n:
Divide and conquer solutions
Lookup in a balanced search tree

Linear n:
When each input element must be processed once.

n log n:
Each input element processed once and processing involves other elements too, for example, sorting.

n^2, n^3:
Quadratic, cubic. Processing all pairs (triples) of elements.

2^n:
Exponential. Processing all subsets of elements.

Summarising Reasoning with Big-Oh

$$O(f(n)) + O(g(n)) = \max\{O(f(n)), O(g(n))\}$$

$$c \cdot O(f(n)) = O(f(n))$$

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

SORTING ALGORITHMS

SELECTION SORT

$$\underbrace{A_0 \le A_1 \le \cdots \le A_{i-1}}_{\text{in their final positions}} \mid \overbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}^{\text{the last } n-i \text{ elements}}$$

After $n - 1$ passes, the list is sorted.
Here is pseudocode of this algorithm, which, for simplicity, list is implemented as an array:

ALGORITHM *SelectionSort*$(A[0..n-1])$
//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  $min \leftarrow i$
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] < A[min]$ $min \leftarrow j$
  swap $A[i]$ and $A[min]$

89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

STARTEGY: **BRUTE FORCE**
STABLE: **NO** **IN-PLACE**
COMPLEXITY: **Θ (n^2)**
WORST BEST CASE: **O (n^2)** **(all inputs)**
NUMBER OF SWAMPS: **Θ (n)** **(n - 1, exactly)**

BUBBLE SORT

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \leq \cdots \leq A_{n-1}}_{\text{in their final positions}}$$

Here is pseudocode of this algorithm.

ALGORITHM *BubbleSort*$(A[0..n-1])$
//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

89 ? 45 68 90 29 34 17
45 89 ? 68 90 29 34 17
45 68 89 ? 90 ? 29 34 17
45 68 89 29 90 ? 34 17
45 68 89 29 34 90 ? 17
45 68 89 29 34 17 | 90

45 ? 68 ? 89 ? 29 34 17 | 90
45 68 29 89 ? 34 17 | 90
45 68 29 34 89 ? 17 | 90
45 68 29 34 17 | 89 90

etc.

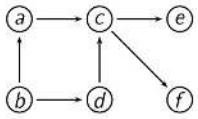
STARTEGY: **BRUTE FORCE**
STABLE: **YES** **IN-PLACE**
COMPLEXITY: **Θ (n^2)**
WORST BEST CASE: **O (n^2)** **(all inputs)**
NUMBER OF SWAMPS: Θ (n^2)

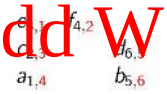
TOPOLOGICAL SORT

DFS application Graph has to be a **DAG**

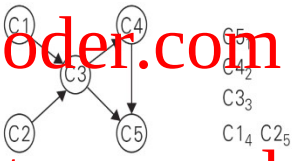
Then we should try to traverse the graph, that is, order the nodes in such a sequence $v_1, v_2, \ldots, v_n$ such that for each edge $(v_i, v_j) \in E$, we have $i < j$.

Using the DFS method and resolving ties by using alphabetical order, the graph gives rise to the traversal stack shown on the right (the popping order shown in red):





Taking the nodes in reverse popping order yields b, d, a, c, f, e.



C5
$C4_2$
$C3_3$
$C1_4$ $C2_5$

(a) (b)

The popping-off order:
C5, C4, C3, C1, C2
The topologically sorted list:
C2 C1→C3→C4→C5

(c)

FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

Another approach can be with **Decrease and conquer** selecting a random **source** in the graph (that is, a node with no incoming edges), list it, and remove it from the graph.

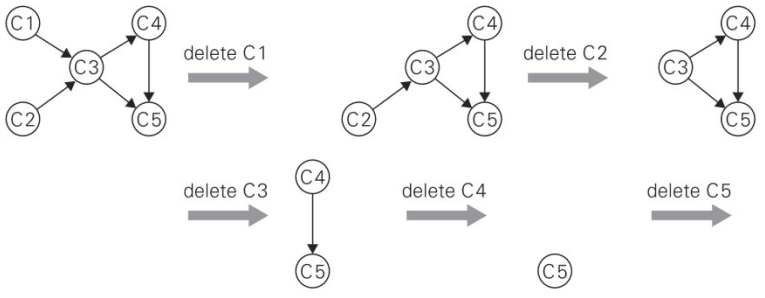


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

INSERTION SORT very good for small arrays (couple of hundred elements)

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
```

$$A[0] \le \cdots \le A[j] < A[j+1] \le \cdots \le A[i-1] \mid A[i] \cdots A[n-1]$$

smaller than or equal to $A[i]$ — greater than $A[i]$

89	**45**	68	90	29	34	17
45	89	**68**	90	29	34	17
45	68	89	**90**	29	34	17
45	68	89	90	**29**	34	17
29	45	68	89	90	**34**	17
29	34	45	68	89	90	**17**
17	29	34	45	68	89	90

STRATEGY: **DECREASE AND CONQUER (DECREASE BY ONE)**
STABLE: **YES** **IN-PLACE**
COMPLEXITY:
WORST CASE: **Θ (n^2)** **(array in decreasing order)**
BEST CASE: **O (n)** **(array in increasing order)** **almost sorted input**
AVERAGE CASE **$n^2 / 4 \in \Theta(n^2)$**

SHELLSORT Very good on medium-sized arrays (up to size 10,000 or so)

Based on INSERTION SORT

- Applies **insertion sort** to each of several **interleaving sublists** of a given list.
- On each pass through the list, the sublists in question are formed by stepping through the list with an increment h_i taken from some predefined decreasing sequence of step sizes, $h_1 > \ldots > h_i > \ldots > 1$, which must end with 1. (The algorithm works for any such sequence, though some sequences are known to yield a better efficiency than others. For example, the sequence 1, 4, 13, 40, 121 . . . used, of course, in **reverse**, is known to be among the best for this purpose.)

STABLE: **NO** **IN-PLACE**
COMPARISONS **O (n√n)** **O ($n^{1.25}$)**

MERGE SORT

also can be performed bottom-up (first work in pairs an so on)

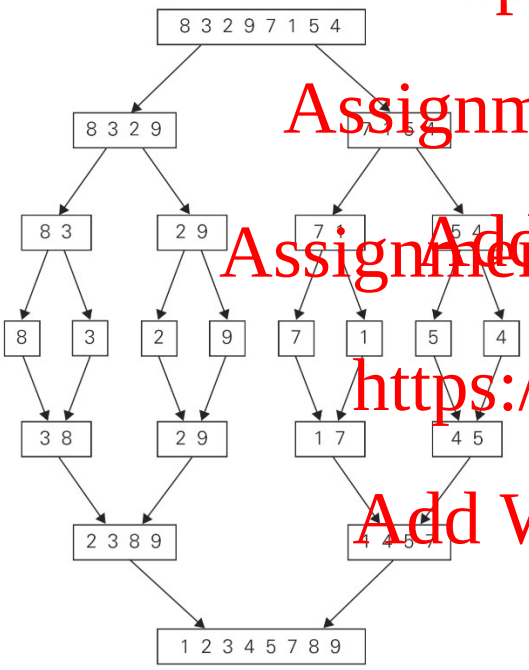
Divides according to a position

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A) //see below

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
if $B[i] \le C[j]$
$A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
$k \leftarrow k + 1$
if $i = p$
copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$



STRATEGY: DIVIDE AND CONQUER — **LINEAR EXTRA STORAGE**

STABLE: YES — **NOT IN-PLACE**

ADVANTAGE over QUICKSORT AND HEAPSORT — **DISADVANTAGE**

COMPLEXITY:

WORST CASE: Θ (n log n) MT small elements in both sides f (n) = n - 1

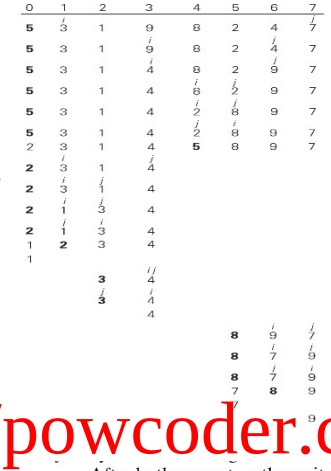
Exactly: $n \log_2 n - n + 1$

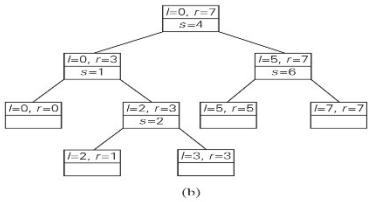
BEST CASE: based in merging time complexity

AVERAGE CASE 0.25n less than WORST CASE still Θ (n log n)

QUICK SORT THE BEST

Divides according to a value (use **PARTITIONS**) Hoare





ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
$s \leftarrow$ *Partition*$(A[l..r])$ //s is a split position
Quicksort$(A[l..s-1])$
Quicksort$(A[s+1..r])$

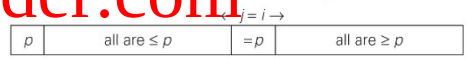
ALGORITHM *HoarePartition*$(A[l..r])$

//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j, respectively:

If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the subarray after exchanging the pivot with $A[j]$:

Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, we have the subarray partitioned, with the split position $s = i = j$:



We can combine the last case with the case of crossed-over indices $(i > j)$ by exchanging the pivot with $A[j]$ whenever $i \geq j$.

STRATEGY: **DIVIDE AND CONQUER**

STABLE: **NO** **IN-PLACE**

DISADVANTAGE over MERGESORT

COMPLEXITY:

WORST CASE: **Θ (n^2)** **for increasing arrays (partitions are size 1 and n-1)**

BEST CASE: **Θ ($n \log_2 n$) MT** **Based in partitioning f (n) = n (best)**

AVERAGE CASE **$2n \ln n \approx 1.39\, n \log_2 n$** **Only 39% more than Best case**

ADVANTAGE **(FASTER THAN MERGESORT AND HEAPSORT)**

IMPROVEMENTS

- better pivot selection methods such as ***randomized quicksort*** that uses a random element or the ***median-of-three*** method that uses the median of the leftmost, rightmost, and the middle element of the array
- switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array
- modifications of the partitioning algorithm such as the three-way partition into segments smaller than, equal to, and larger than the pivot (see Problem 9 in this section's exercises)

function SORT$(A[lo..hi])$
QUICKSORT$(A[lo..hi])$
INSERTIONSORT$(A[lo..hi])$

function QUICKSORT$(A[lo..hi])$
if $lo + 10 < hi$ **then**
$s \leftarrow$ PARTITION$(A[lo..hi])$
QUICKSORT$(A[lo..s-1])$
QUICKSORT$(A[s+1..hi])$

HEAP SORT

Given unsorted array $H[1..n]$:

Step 1 Turn H into a heap.

Step 2 Apply the eject operation $n - 1$ times.

Create a **heap** (by bottom up)		O(n)
Eject operation (n – 1 times)	(n – 1)log n	O(n log n)

STRATEGY: **TRANSFORM AND CONQUER**

STABLE: **NO** **IN-PLACE**

DISADVANTAGE over MERGESORT

COMPLEXITY AVERAGE AND WORST: **Θ (n log n)**

AVERAGE **Slower than QUICKSORT, but stronger performance guarantee**

SORTING BY COUNTING

COMPARISON SORTING COUNT

ALGORITHM *ComparisonCountingSort*$(A[0..n-1])$

```
//Sorts an array by comparison counting
//Input: An array A[0..n − 1] of orderable elements
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for i ← 0 to n − 1 do Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do S[Count[i]] ← A[i]
return S
```


Array $A[0..5]$

62	31	84	96	19	47

Initially	*Count* []	0	0	0	0	0	0
After pass i = 0	*Count* []		1	0	0	0	0
After pass i = 1	*Count* []			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

STRATEGY: **SPACE-TIME TRADE-OFF (INPUT ENHACEMENT)**

STABLE: **YES** **NO IN-PLACE**

COMPLEXITY **Θ (n^2)** **Extra linear time**

DISTRIBUTION COUNTING

13	11	12	13	12	12

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

ALGORITHM *DistributionCountingSort*$(A[0..n-1], l, u)$

```
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n − 1] of integers between l and u (l ≤ u)
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for j ← 0 to u − l do D[j] ← 0                          //initialize frequencies
for i ← 0 to n − 1 do D[A[i] − l] ← D[A[i] − l] + 1     //compute frequencies
for j ← 1 to u − l do D[j] ← D[j − 1] + D[j]            //reuse for distribution
for i ← n − 1 downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← D[j] − 1
return S
```

	D[0..2]			*S*[0..5]					
A [5] = 12	1	**4**	6				12		
A [4] = 12	1	**3**	6			12			
A [3] = 13	1	2	**6**						13
A [2] = 12	1	**2**	5		12				
A [1] = 11	**1**	1	5	11					
A [0] = 13	0	1	**5**					13	

STRATEGY: SPACE-TIME TRADE-OFF (INPUT ENHACEMENT)
STABLE: YES **NO IN-PLACE**
COMPLEXITY **Θ (n)** **exactly 2n**
But for arrays with a range of values fixed (111000102220022220111)

SEARCHING ALGORITHMS

SEQUENTIAL SEARCH

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$
//Searches for a given value in a given array by sequential search
//Input: An array $A[0..n-1]$ and a search key K
//Output: The index of the first element in A that matches K
// or -1 if there are no matching elements
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
$i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

ALGORITHM *SequentialSearch2*$(A[0..n], K)$
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in $A[0..n-1]$ whose value is
// equal to K or -1 if no such element is found
$A[n] \leftarrow K$
$i \leftarrow 0$
while $A[i] \neq K$ **do**
$i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

STRATEGY: BRUTE FORCE
COMPLEXITY: **O (n)**
BEST CASE: **O (1)**
WORST CASE: **O (n)**
AVERAGE CASE: **O ((n+1)/2)**

BINARY SEARCH

MUST BE IN A SORTED ARRAY **Better for smaller files**

function BINSEARCH$(A[], lo, hi, key)$
if $lo > hi$ **then return** -1
$mid \leftarrow lo + (hi - lo)/2$
if $A[mid] = key$ **then return** mid
else
if $A[mid] > key$ **then**
return BINSEARCH$(A, lo, mid - 1, key)$
else return BINSEARCH$(A, mid + 1, hi, key)$

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
$l \leftarrow 0; \quad r \leftarrow n - 1$
while $l \leq r$ **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

STRATEGY: DECREASE AND CONQUER (by a constant factor)
COMPLEXITY: **O (log n)**
WORST CASE (no K in the array) **Θ (log n)**
AVERAGE CASE **Θ ($\log_2$ n)**

INTERPOLATION SEARCH AND HASHING HAS A BETTER AVERAGE-CASE TIME EFFICIENCY

INTERPOLATION SEARCH

(in a sorted array) that increase linearly **Better for larger files**

$$m \leftarrow lo + \left\lfloor \frac{k - A[lo]}{A[hi] - A[lo]}(hi - lo) \right\rfloor$$

round-off

STRATEGY: DECREASE AND CONQUER (variable size decrease)
AVERAGE CASE O (log log n)
WORST CASE O (n) (bad performance)

SEARCHING AND INSERTION IN A BINARY SEARCH TREE

STRATEGY: DECREASE AND CONQUER (variable size decrease)
AVERAGE CASE Θ (log n) precisely $2 \ln n \approx 1.39 \log_2(n)$
WORST CASE Θ (n) (bad performance)

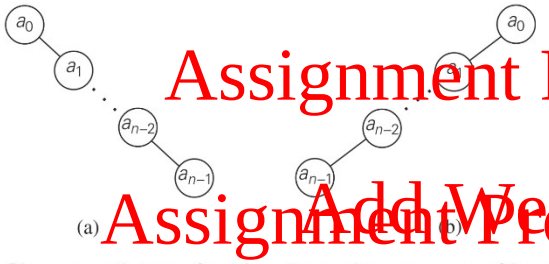


FIGURE 4.15 Binary search trees for (a) an increasing sequence of keys and (b) a decreasing sequence of keys.

WORST CASES

STRING MATCHING ALGORITHMS

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

STRATEGY: BRUTE FORCE
COMPLEXITY: O (nm)
BEST CASE: O (m)
WORST CASE: O (nm) number of trials (n-m+1)*m
AVERAGE CASE: O (n + m)

HORSPOOL'S ALGORITHM

FOUR CASES

ALGORITHM *ShiftTable*$(P[0..m-1])$

```
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P[0..m − 1] and an alphabet of possible characters
//Output: Table[0..size − 1] indexed by the alphabet's characters and filled with
//        shift sizes computed by formula (7.1)
for i ← 0 to size − 1 do Table[i] ← m
for j ← 0 to m − 2 do Table[P[j]] ← m − 1 − j
return Table
```

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \quad \textbf{(7.1)}$$

For example, for the pattern BARBER, all the table's entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively.

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//        or −1 if there are no matches
ShiftTable(P[0..m − 1])          //generate Table of shifts
i ← m − 1                        //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                        //number of matched characters
    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
```

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
```

STRATEGY: **SPACE-TIME TRADE-OFF (INPUT ENHACEMENT)**

COMPLEXITY:		**Θ (n)**	
WORST CASE	**like Brute Force**	**O (nm)**	**the first char is different in a text of 2 char**
AVERAGE CASE		**Θ (n)**	**00000000000000000**
			100000
BOYER MOORE ALGORITHM		O (n + m)	more sophisticated shifting strategy

ANALYSIS OF RECURSIVE ALGORITHMS

Telescoping

The recursive equation was:

$$\boxed{M(n) = M(n-1) + 1 \quad (\text{for } n > 0)}$$

Use the fact $M(n-1) = M(n-2) + 1$ to expand the right-hand side:

$$M(n) = [M(n-2) + 1] + 1 = M(n-2) + 2$$

and keep going:

$$\ldots = [M(n-3) + 1] + 2 = M(n-3) + 3 = \ldots = M(n-n) + n = n$$

where we used the base case $M(0) = 0$ to finish.

Establishing Growth Rate

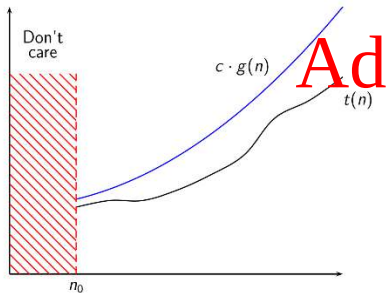
A more common approach uses

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies } t \text{ grows asymptotically slower than } g \\ c & \text{implies } t \text{ and } g \text{ have same order of growth} \\ \infty & \text{implies } t \text{ grows asymptotically faster than } g \end{cases}$$

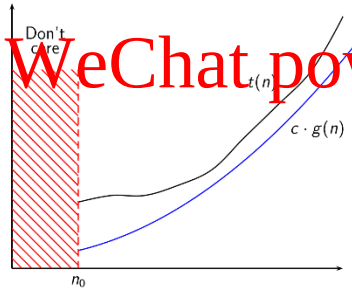
$$\begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.

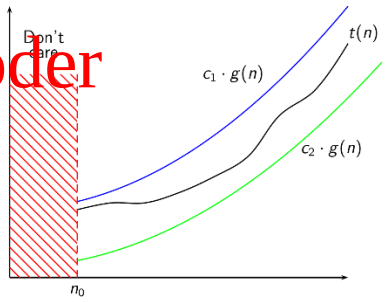
Big-Oh: What $t(n) \in O(g(n))$ Means



Big-Omega: What $t(n) \in \Omega(g(n))$ Means



Big-Theta: What $t(n) \in \Theta(g(n))$ Means



Some Useful Formulas

From Stirling's formula:

$$n! = O(n^{n+\frac{1}{2}})$$

Some useful sums:

$$\begin{aligned} \textstyle\sum_{i=0}^{n} i^2 &= \tfrac{n}{3}(n+\tfrac{1}{2})(n+1) \\ \textstyle\sum_{i=0}^{n}(2i+1) &= (n+1)^2 \\ \textstyle\sum_{i=1}^{n} 1/i &= O(\log n) \end{aligned}$$

L'Hôpital's Rule

Often it is helpful to use L'Hôpital's rule:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

where t' and g' are the derivatives of t and g.

For example, we can show that $\log_2 n$ grows slower than $\sqrt{n}$:

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

BRUTE FORCE ALGORITHMS

- Selection sort
- String matching
- Closest pair (better divide and conquer)
- Exhaustive search for combinatorial solutions
- Graph traversal

PROBLEMS:
Travelling Salesperson (TSP) (Hamiltonian (nodes), Eulerian (edges) circuit)
Knapsack (better dynamic programming)
Assignment problem

ALGORITHM *BruteForceClosestPair*(P)

//Finds distance between two closest points in the plane by brute force
//Input: A list P of n $(n \geq 2)$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$
//Output: The distance between the closest pair of points
$d \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //*sqrt* is square root
return d

STRATEGY: **BRUTE FORCE**
COMPLEXITY: **$O(n^2)$**

FIBONACCI (better way $O(\log n)$)

function FIB(n)
  if $n = 0$ **then**
    return 1
  if $n = 1$ **then**
    return 1
  return FIB$(n - 1) +$ FIB$(n - 2)$

EXPONENTIAL

function FIB(n, a, b)
  if $n = 0$ **then**
    return a
  return FIB$(n - 1, a + b, a)$

Initial call: FIB$(n, 1, 0)$

function FIB(n)
  $a \leftarrow 1$
  $b \leftarrow 0$
  while $n > 0$ **do**
    $a \leftarrow a + b$
    $b \leftarrow a - b$
    $n \leftarrow n - 1$
  return a

LINEAR

GRAPHS

More Graph Concepts: Degrees of Nodes

If $(v, u) \in E$ then v and u are adjacent, or neighbours.

(v, u) is incident on, or connects, v and u.

The degree of node v is the number of edges incident on v.

For directed graphs, we talk about v's in-degree (number of edges going to v) and its out-degree (number of edges going from v).

A path in $\langle V, E \rangle$ is a sequence of nodes $v_0, v_1, \ldots, v_k$ from V, so that $(v_i, v_{i+1}) \in E$.

The path $v_0, v_1, \ldots, v_k$ has length k.

A simple path is one that has no repeated nodes.

A cycle is a simple path, except that $v_0 = v_k$, that is, the last node is the same as the first node.

Graph Representation

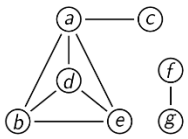
Adjacency matrix and Adjacency list

(1) The **adjacency matrix** of a directed graph does not have to be symmetric;
(2) An edge in a directed graph has just one (not two) corresponding nodes in the digraph's **adjacency lists**.

GRAPH TRAVERSAL

DFS Stack discipline **BACKTRACKING**

Depth-First Search: The Traversal Stack



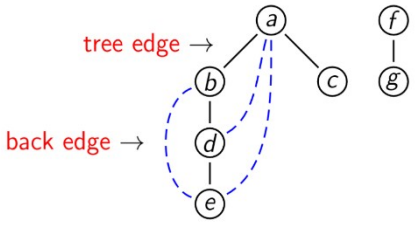
Levitin uses a more compact notation for the stack's history. Here is how the stack develops, in Levitin's notation:

$e_{4,1}$
$d_{3,2}$
$b_{2,3}$ $c_{5,4}$ $g_{7,6}$
$a_{1,5}$ $f_{6,7}$

The first subscripts give the order in which nodes are pushed, the second the order in which they are popped off the stack.

Another useful tool for depicting a DF traversal is the DFS tree (for a connected graph).

More generally, we get a DFS forest:



Depth-First Search: The Algorithm

```
function DFS(⟨V, E⟩)
    mark each node in V with 0
    count ← 0
    for each v in V do
        if v is marked 0 then
            DfsExplore(v)

function DfsExplore(v)
    count ← count + 1
    mark v with count
    for each node w adjacent to v do
        if w is marked with 0 then
            DfsExplore(w)
```

The "marking" of nodes is usually done by maintaining a separate array, `mark`, indexed by V.

For example, when we wrote "mark v with *count*", that would be implemented as "`mark[v] := count`".

How to find the nodes adjacent to v depends on the graph representation used.

Using an adjacency matrix `adj`, we need to consider `adj[v,w]` for each `w` in V. Here the complexity of graph traversal is $\Theta(|V|^2)$.

Using adjacency lists, for each v, we traverse the list `adj[v]`.

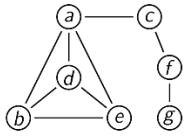
In this case, the complexity of traversal is $\Theta(|V| + |E|)$. Why? ∎

- To check if a graph is **connected**.
- To check if a graph has a **cycle**.
- Is a **DAG** if there is not back edge in the DFS forest.
- Find **articulation point**

BFS Queue discipline **CONCENTRIC**

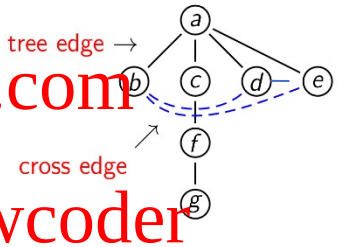
Breadth-First Search: The Traversal Queue

BFS uses a queue discipline by keeping V in a queue. Here is the BFS traversal for the example: track of pending tasks.



How the queue develops for the example:

a_1			
b_2	c_3	d_4	e_5
c_3	d_4	e_5	
d_4	e_5	f_6	
e_5	f_6		
f_6			
g_7			



The subscript again is Levitin's; it gives the order in which nodes are processed.

In general, we may get a BFS forest.

Breadth-First Search: The Algorithm

```
function BFS(⟨V, E⟩)
    mark each node in V with 0
    count ← 0
    for each v in V do
        if v is marked 0 then
            count ← count + 1
            mark v with count
            queue ← [v]                    // queue containing just v
            while queue is non-empty do
                u ← eject(queue)           // dequeues u
                for each node w adjacent to u do
                    if w is marked with 0 then
                        count ← count + 1
                        mark w with count
                        inject(queue, w)   // enqueues w
```

The "marking" of nodes is usually done by maintaining a separate array, `mark`, indexed by V.

For example, when we wrote "mark v with *count*", that would be implemented as "`mark[v] := count`".

How to find the nodes adjacent to v depends on the graph representation used.

Using an adjacency matrix `adj`, we need to consider `adj[v,w]` for each `w` in V. Here the complexity of graph traversal is $\Theta(|V|^2)$.

Using adjacency lists, for each v, we traverse the list `adj[v]`.

In this case, the complexity of traversal is $\Theta(|V| + |E|)$. Why? ∎

- **Shortest path** between two nodes.
- To check if a graph is **connected**.
- To check if a graph has a **cycle**. (DFS is better)
- NOT GOOD to find **articulation point**

	DFS	BFS
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

DECREASE AND CONQUER

The **bottom-up** variation is usually implemented iteratively, starting with a solution to the **smallest instance** of the problem; it is called sometimes the **incremental approach**

- Decrease by a constant (one)

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0 \\ 1 & \text{if } n = 0, \end{cases}$$

  - **INSERTION SORT (Decrease by one)**
  - **TOPOLOGICAL SORT (source with no coming edges)**
  - **ALGORITHMS FOR COMBINATORIAL OBJECTS**

- Decrease by a constant factor (two) Complexity: **Θ (log n)**

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

  - **BINARY SEARCH**
  - **FAKE COIN PROBLEM** $W(n) = \lfloor \log_2 n \rfloor$ **, dividing on 3 piles, complexity is less, though $\log_3 n$**
  - **RUSSIAN PEASANT MULTIPLICATION**

- $$n \cdot m = \frac{n}{2} \cdot 2m. \qquad n \cdot m = \frac{n-1}{2} \cdot 2m + m.$$

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1040	
1	2080	(+1040)
	2080	+(130 + 1040) = 3250

(a)

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

(b)

FIGURE 4.11 Computing 50 · 65 by the Russian peasant method.

  - **JOSEPHUS PROBLEM**

- Variable size decrease

  gcd (m, n) = gcd (n, m mod n)

  - **QUICKSELECT:** COMPUTING A MEDIAN, THE SELECTION PROBLEM (find de k[th] smallest element)

    Pivot, **PARTITIONS** (Lomuto)

ALGORITHM *LomutoPartition*$(A[l..r])$

```
//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray A[l..r] of array A[0..n − 1], defined by its left and right
//       indices l and r (l ≤ r)
//Output: Partition of A[l..r] and the new position of the pivot
p ← A[l]
s ← l
for i ← l + 1 to r do
    if A[i] < p
        s ← s + 1;  swap(A[s], A[i])
swap(A[l], A[s])
return s
```


ALGORITHM *Quickselect*$(A[l..r], k)$

```
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray A[l..r] of array A[0..n − 1] of orderable elements and
//       integer k (1 ≤ k ≤ r − l + 1)
//Output: The value of the kth smallest element in A[l..r]
s ← LomutoPartition(A[l..r]) //or another partition algorithm
if s = k − 1 return A[s]
else if s > k − 1 Quickselect(A[l..s − 1], k)
else Quickselect(A[s + 1..r], k − s)
```


0	1	2	3	4	5	6	7	8
s	*i*							
4	1	10	8	7	12	9	2	15
	s	*i*						
4	1	10	8	7	12	9	2	15
	s						*i*	
4	1	10	8	7	12	9	2	15
		s					*j*	
4	1	2	8	7	12	9	10	15
		s					*j*	
4	1	2	8	7	12	9	10	15
2	1	**4**	8	7	12	9	10	15

Since $s = 2$ is smaller than $k - 1 = 4$, we proceed with the right part of the array:

0	1	2	3	4	5	6	7	8
			s	*i*				
8	7	12	9	10	15			
			s	*i*				
8	7	12	9	10	15			
			s				*i*	
8	7	12	9	10	15			
7	**8**	12	9	10	15			

Now $s = k - 1 = 4$, and hence we can stop: the found median is 8, which is greater than 2, 1, 4, and 7 but smaller than 12, 9, 10, and 15. ■

COMPLEXITY:

BEST CASE:	**O (n)**	
WORST CASE:	**O (n^2)**	**increasing array INPUT**
AVERAGE CASE:	**O (n)**	**surprisely**

  - **INTERPOLATION SEARCH**
  - **SEARCHING AND INSERTION IN A BINARY SEARCH TREE**

DIVIDE AND CONQUER

BEST-KNOWN GENERAL ALGORITHMIC DESIGN

Divided a problem (size n) in two sub problems (size n/2) and then merge its results

More general sub problems of size n/b, with a of them needing to be solved (n is power of b)

EXAMPLE: SUM OF LIST OF NUMBERS

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1}).$$

lem's instance of size n is divided into two instances of size $n/2$. More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.) Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = aT(n/b) + f(n), \tag{5.1}$$

where $f(n)$ is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions. (For the sum example above, $a = b = 2$ and $f(n) = 1$.) Recurrence (5.1) is called the ***general divide-and-conquer recurrence***. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem (see Appendix B).

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

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

Analogous results hold for the O and Ω notations, too.

For example, the recurrence for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

- **MERGESORT**
- **QUICKSORT (HOARE PARTITION) PIVOT IMPROVEMENTS**
- **CLOSEST PAIR REVISITED** **O (n log n)**
- **BINARY TREE TRAVERSALS**

ALGORITHM *Height*(T)

```
//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if T = ∅ return −1
else return max{Height(T_left), Height(T_right)} + 1
```

We measure the problem's instance size by the number of nodes $n(T)$ in a given binary tree T. Obviously, the number of comparisons made to compute the maximum of two numbers and the number of additions $A(n(T))$ made by the algorithm are the same. We have the following recurrence relation for $A(n(T))$:

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

COMPLEXITY

C (n) = n + x = 2n + 1 **(Number of comparisons to check whether the tree is empty)**

A (n) = n **(Number of additions)**

x = n + 1 **(x = external nodes (leaves), n = internal nodes (parental))**

2n + 1 = x + n

Preorder traversal visits the root, then the left subtree, and finally the right subtree.

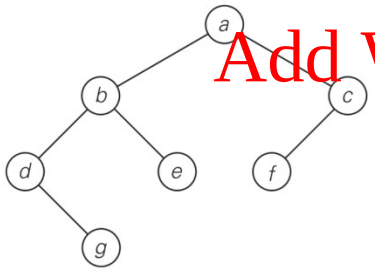
Inorder traversal visits the left subtree, then the root, and finally the right subtree.

Postorder traversal visits the left subtree, the right subtree, and finally the root.

Level-order traversal visits the nodes, level by level, starting from the root.

Can be made as Stack

IN ORDER TRAVERSAL IN A BST PRODUCES ELEMENTS IN SORTED ORDER



preorder: *a, b, d, g, e, c, f*
inorder: *d, g, b, e, a, f, c*
postorder: *g, d, e, b, f, c, a*

function PREORDERTRAVERSE(T)
  if T is non-empty **then**
    visit T_{root}
    PREORDERTRAVERSE(T_{left})
    PREORDERTRAVERSE(T_{right})

.6 Binary tree and its traversals.

COMPLEXITY

A (n) = n **(Number of additions)**

TRANSFORM AND CONQUER

- Transformation to a simpler or more convenient instance of the same problem—we call it ***instance simplification***.
- Transformation to a different representation of the same instance—we call it ***representation change***.
- Transformation to an instance of a different problem for which an algorithm is already available—we call it ***problem reduction***.

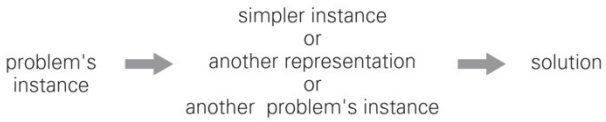


FIGURE 6.1 Transform-and-conquer strategy.

INSTANCE SIMPLIFICATION

PRESORTING With MERGESORT OR QUICKSORT (n log n)

So far, we have discussed three elementary sorting algorithms—selection sort, bubble sort, and insertion sort—that are quadratic in the worst and average cases, and two advanced algorithms—mergesort, which is always in $\Theta(n \log n)$, and quicksort, whose efficiency is also $\Theta(n \log n)$ in the average case but is quadratic in the worst case. Are there faster sorting algorithms? As we have already stated in

- **CHECKING ELEMENT UNIQUENESS IN AN ARRAY** There are a Brute force of $O(n^2)$

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$
```
//Solves the element uniqueness problem by sorting the array first
//Input: An array A[0..n − 1] of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
sort the array A
for i ← 0 to n − 2 do
    if A[i] = A[i + 1] return false
return true
```

COMPLEXITY Θ (n log n)
Based in SORTING time

- **COMPUTING A MODE** There are a Brute force of $O(n^2)$

ALGORITHM *PresortMode*$(A[0..n-1])$
```
//Computes the mode of an array by sorting it first
//Input: An array A[0..n − 1] of orderable elements
//Output: The array's mode
sort the array A
i ← 0                          //current run begins at position i
modefrequency ← 0              //highest frequency seen so far
while i ≤ n − 1 do
    runlength ← 1; runvalue ← A[i]
    while i + runlength ≤ n − 1 and A[i + runlength] = runvalue
        runlength ← runlength + 1
    if runlength > modefrequency
        modefrequency ← runlength; modevalue ← runvalue
    i ← i + runlength
return modevalue
```

COMPLEXITY Θ (n log n)
Based in SORTING time

- **SEARCHING PROBLEM**

$$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

Sequential search has O (n) and it is better
But presorting and searching more elements (by Binary search), each one cost only O (log n)
HOW MUCH SEARCHES JUSTIFY A PRESORTING??

n log n + log n + log n + log n +…. **(n + a)*log n**
n + n + n +…. **a*n**

When (n + a)*log n <= a*n

- **LONGEST AND SHORTEST PATH IN DIGRAPH**

Presorting with TOPOLOGICAL SORT **Source (Decrease and conquer) (linear)) (|V|)**
DFS (brute force) (quadratic or .) ($|V|^2$) or (|V| + |E|)

REPRESENTATION CHANGE

- **TRANSFORM A SET OR ARRAY INTO A BINARY SEARCH TREE**
  - Advantage in Searching, insertion and deletion O (log n)
  - In worst case is O (n) (in unbalanced trees)
  - Avoid this worst cases with balanced tree

INSTANCE SIMPLIFICATION

An unbalanced tree is transform to a balanced one **(AVL tree)**

REPRESENTATION CHANGE

Create a 2-3 trees, 2-3-4 trees: **B-trees**

AVL TREES

is a binary search tree but balanced

Nodes' balance factors just can be **-1, 0 or +1** (difference between **L sub tree Height** and **R sub H**)
Leaves has a balance factor = 0 Height of empty tree is -1
2 = R or LR
-2 = L or RL

Rotations made in the closest (lowest) node inserted

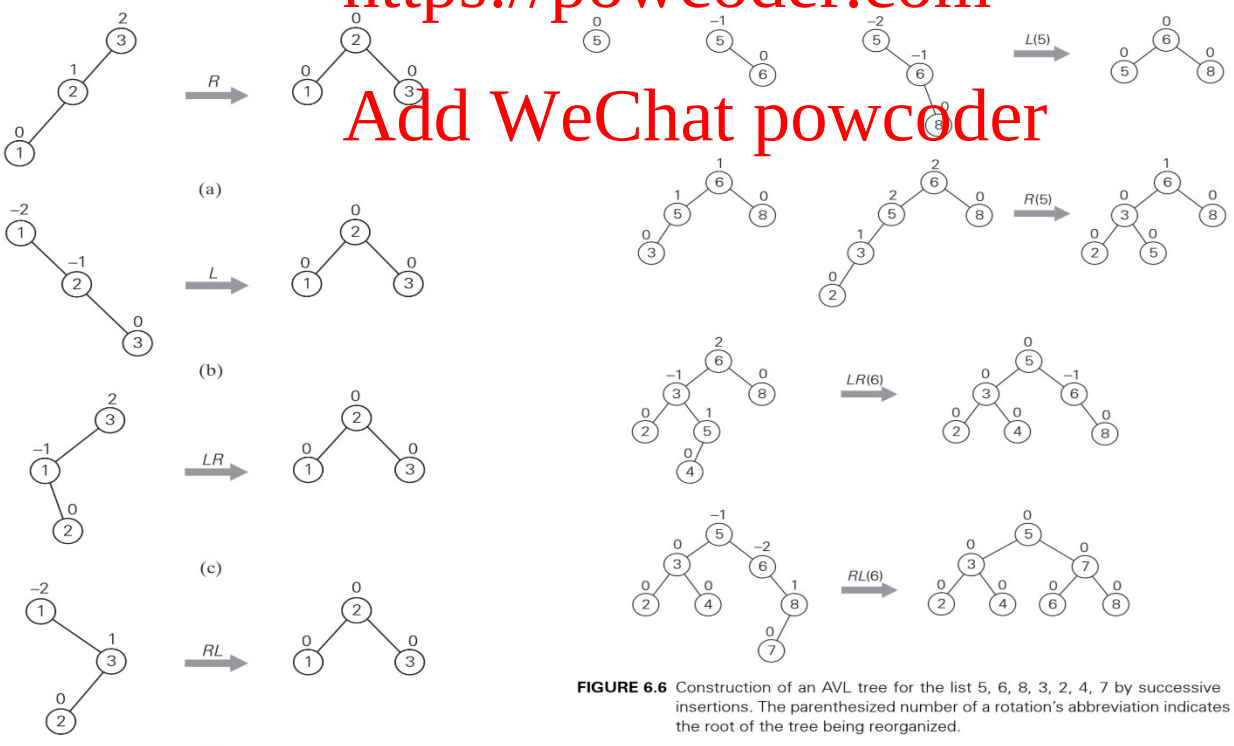


FIGURE 6.6 Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

$$\lfloor \log_2 n \rfloor \le h < 1.4405 \log_2(n+2) - 1.3277$$

log n

INSERTION, SEARCHING and deletion
COMPLEXITY WORST AVERAGE CASE Θ (log n)

2-3 TREES

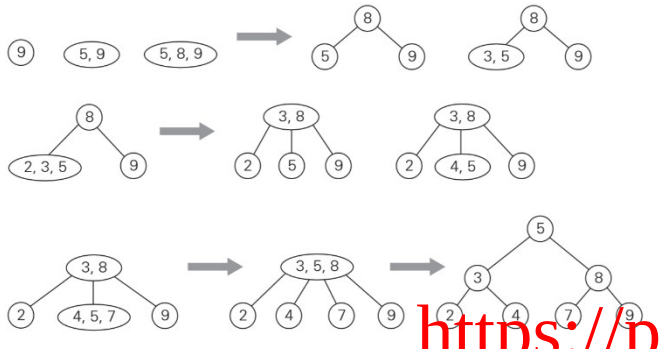


FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

INSERTION, SEARCHING and deletion
COMPLEXITY WORST AVERAGE CASE Θ (log n)

HEAPS

just a COMPLETE binary tree (not BST)

Applications: Prim's algorithm, Dijkstra's algorithm, Huffman encoding, and branch-and-bound applications.

The Priority Queue

As an abstract data type, the priority queue supports the following operations on a "pool" of elements (ordered by some linear order):

- find an item with maximal priority
- insert a new item with associated priority
- test whether a priority queue is empty
- eject the largest element

Other operations may be relevant, for example:

- replace the maximal item with some new item
- construct a priority queue from a list of items
- join two priority queues

Special instances are obtained when we use time for priority:

- If "large" means "late" we obtain the stack.
- If "large" means "early" we obtain the queue.

Possible Implementations of the Priority Queue

Assume priority = key.

	INJECT(e)	EJECT()
Unsorted array or list		
Sorted array or list		
Heap	$O(\log n)$	$O(\log n)$

The Heap

A heap is a complete binary tree which satisfies the heap condition:

Each child has a priority (key) which is no greater than its parent's.

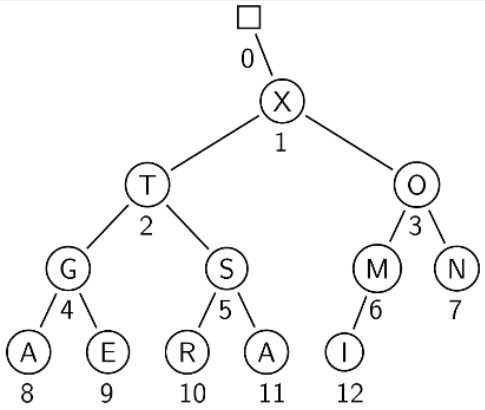
This guarantees the the root of the tree is a maximal element.

(Sometimes we talk about this as a max-heap—one can equally well have min-heaps, in which each child is no smaller than its parent.)

Heaps as Arrays

We can utilise the completeness of the tree and place its elements in level-order in an array H.

Note that the children of node i will be nodes $2i$ and $2i+1$.



H:

	X	T	O	G	S	M	N	A	E	R	A	I
0	1	2	3	4	5	6	7	8	9	10	11	12

This way, the heap condition is very simple:

For all $i \in \{0, 1, \ldots, n\}$, we must have $H[i] \leq H[\lfloor i/2 \rfloor]$.

Properties of the Heap

The root of the tree $H[1]$ holds a maximal item; the cost of EJECT is $O(1)$ plus time to restore the heap.

The height of the heap is $\lfloor \log_2 n \rfloor$.

Each subtree is also a heap.

The children of node i are $2i$ and $2i+1$.

The nodes which happen to be parents are in array positions 1 to $\lfloor n/2 \rfloor$.

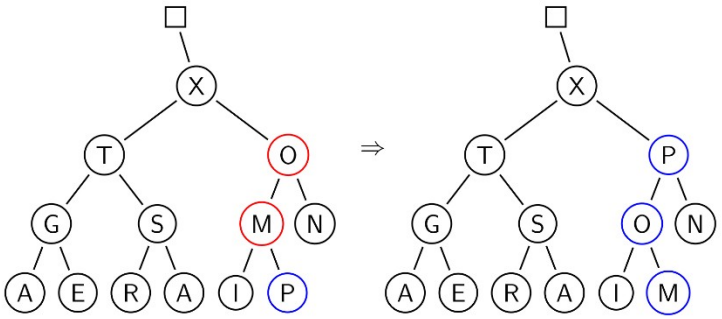
It is easier to understand the heap operations if we think of the heap as a tree.

the array representation

index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1
		parents					leaves				

Injecting a New Item

Place the new item at the end; then let it "climb up", repeatedly swapping with parents that are smaller:

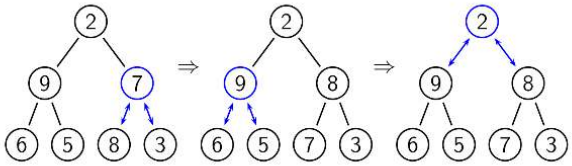


JUST INJECTING O(log n)

CREATION BY INJECTING (CLIMB UP): Complexity O(n log n)

Building a Heap Bottom-Up

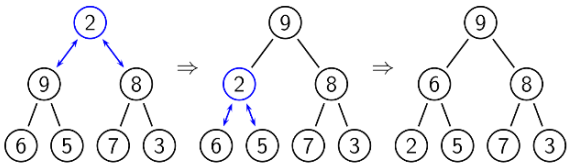
To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. The construction cost will be $n \log n$. But there is a better way:



Start with the last parent and move backwards, in level-order. For each parent node, if the largest child is larger than the parent, swap it with the parent.

Building a Heap Bottom-Up: Sifting Down

Whenever a parent is found to be out of order, let it "sift down" until both children are smaller:



Algorithm to Turn $H[1..n]$ into a Heap, Bottom-Up

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i
    v ← H[k]
    heap ← False
    while not heap and 2 × k ≤ n do
        j ← 2 × k
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← True
        else
            H[k] ← H[j]
            k ← j
    H[k] ← v
```

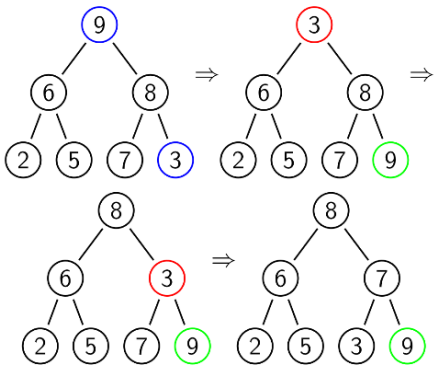

CREATION BY BOTTOM UP: **Complexity O(n)** **fewer than 2n**

Ejecting a Maximal Element from a Heap

Here the idea is to swap the root with the last item z in the heap, and then let z "sift down" to its proper place.

After this, the last element (here shown in green) is no longer considered part of the heap, that is, n is decremented.

Clearly ejection is $O(\log n)$.



EJECTING **O(log n)**

Heapsort

Heapsort is a $\Theta(n \log n)$ sorting algorithm, based on the idea from this exercise.

Given unsorted array $H[1..n]$:

Step 1 Turn H into a heap.

Step 2 Apply the eject operation $n - 1$ times.

TIME/SPACE TRADEOFFS

Graph traversals complexity depends on representation: **n^2** (adjacency matrix) or **n + m** (adjacency list) (n = vertices, m = edges), **one better approach** is by working in a **sparse graph** (fewer edges than vertices), the use of adjacency list is powered ($m \in O(n)$).

TECHNICS

Input enhancement

Preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward.

- **Sorting by counting**
- **Boyer-Moore** algorithm for string matching and its simplified version suggested by **Horspool**

Prestructuring

Uses extra space to facilitate faster and/or more flexible access to the data. This name highlights two facets of this variation of the space-for-time trade-off: **some processing is done before** a problem in question is actually solved but, unlike the input-enhancement variety, it **deals with access structuring**.

- Hashing
- Indexing with B-trees

Dynamic programming

This strategy is based on **recording solutions to overlapping sub problems** of a given problem in a table from which a **solution to the problem in question** is then obtained.

Data compression

HASHING

To implement dictionaries

Based in a hash table (**array H [0…m-1]**) and **hash function** (to get the **hash address**))

For K = non-negative integers **h (K)** = K mod m most common

For K = char **ord(K)** (in the alphabet) and **h (K)** = ord(K) mod m

For K = string $c_0c_1c_2...c_{s-1}$ $(\sum_{i=0}^{s-1} ord(c_i)) \bmod m.$ or $h \leftarrow 0;$ **for** $i \leftarrow 0$ **to** $s-1$ **do** $h \leftarrow (h * C + ord(c_i)) \bmod m,$

C is a constant larger than every *ord(c_i).*

ANOTHER WAY

Hashing of Strings

For simplicity assume A $\mapsto$ 1, B $\mapsto$ 2, etc.
Also assume we have, say, 26 characters, and $m = 101$.

Then we can think of a string as a long binary string:

A K E Y $\mapsto$ 00001010110010111001 (= 44217)

$$44217 \bmod 101 = 80$$

So 80 is the position of string A K E Y in the hash table.

We deliberately chose m to be prime.

$$44217 = 1 \cdot 32^3 + 11 \cdot 32^2 + 5 \cdot 32 + 25$$

With $m = 32$, the hash value of any key is the last character's value!

Hashing of Strings

Even with long strings as keys, we can avoid large numbers in the hash calculations, using Horner's rule. Utilize

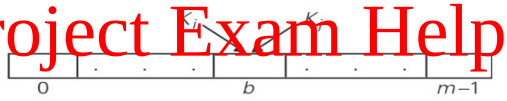
$$\begin{aligned}(x + y) \bmod m &= ((x \bmod m) + (y \bmod m)) \bmod m \\ (x \times y) \bmod m &= ((x \bmod m) \times (y \bmod m)) \bmod m\end{aligned}$$

Naively,

V E R Y L O N G K E Y $\mapsto 22 \times 32^{10} + 5 \times 32^9 + \cdots$

That becomes a very large number. Instead factor out repeatedly:

$$(\cdots ((22 \times 32 + 5) \times 32 + 18) \times 32 + \cdots + 25$$



Collision of two keys in hashing: $h(K_i) = h(K_j)$.

PROBLEMS WITH COLLISSION

WORST CASE: all the keys could be hashed to the same cell of the hash table

APPROPIATELY CHOSEN **HASH TABLE SIZE** AND A **GOOD HASH FUNCTION**

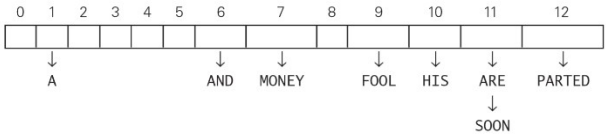
COLLISION RESOLUTION MECHANISM

- **Open hashing** (separate chaining)
- **Closed hashing** (open addressing)

OPEN HASHING (SEPARATE CHAINING)

linked list attached to the cell

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



h(K) = $(\sum_{i=0}^{s-1} ord(c_i)) \bmod m,$ **m = 13**

To search **KID**, use the hash function (= 11), then traverse the linked list comparing the values.

The efficiency of searching depends on the **lengths of the linked lists**, which, in turn, depend on the **dictionary** and **table sizes**, as well as the **quality of the hash function**

Load factor: α = n/m (=1 optimum) n = number of keys m = size of hash table
n < m and **m < n** is possible

$$S \approx 1 + \frac{\alpha}{2} \quad \text{and} \quad U = \alpha$$

S = successful search U = unsuccessful

Compared with sequential search, reduces the number of comparisons by a factor of m
Assumption: keys uniformly distributed

For a SEARCH (optimum) = 1 or 2 comparisons + search (linear) + construction of hash table (once)
COST: EXTRA SPACE

SEARCHING, INSERTION AND DELETION
COMPLEXITY AVERAGE CASE: **θ (1)** **for** **α = n/m = 1** **n = m**

Closed hashing (open addressing) **No linked list**

Only **n < m** is possible

Collision resolution by **Linear Probing**

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

	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

FIGURE 7.6 Example of a hash table construction with linear probing.

Hash function use the next empty cell

Searching: Unsuccessful: if **cell empty**
Successful: if **not empty** and **continue to next cell until** an **empty cell**

Deleting is more complicated

$$S \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \quad \text{and} \quad U \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

α < 0.9 is the worst

PROBLEM: CLUSTERING (between next cells previously occupied)

Collision resolution by **DOUBLE HASHING** another hashing function to fix increments
Superior to Linear probing

$$(l + s(K)) \bmod m, \qquad (l + 2s(K)) \bmod m, \qquad \ldots .$$

For example, we may choose $s(k) = 1 + k \bmod 97$.

By this we mean, if $h(k)$ is occupied, next try $h(k) + s(k)$, then $h(k) + 2s(k)$, and so on.

m must be **prime**

s (k) = m – 2 – k mod (m–2) and s (k) = 8 – (k mod8) **for small tables**
s (k) = k mod 97 + 1 **for larger ones**

The standard approach to avoiding performance deterioration in hashing is to keep track of the load factor and to rehash when it reaches, say, 0.9.

Rehashing means allocating a larger hash table (typically about twice the current size), revisiting each item, calculating its hash address in the new table, and inserting it.

REHASHING

BALANCED SEARCH TREE AND HASHING: COMPARISON

	HASHING	BST
Searching, insertion and deletion		
Best best case	**θ (1)**	**θ (log n)**
But in average	**θ (n)**	average and worst cases
Ordering Preservation	**NO**	**YES**

DYNAMIC PROGRAMMING

COIN ROW PROBLEM

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \quad \text{for } n > 1,$$
$$F(0) = 0, \qquad F(1) = c_1.$$ 8.3

ALGORITHM *CoinRow*$(C[1..n])$

//Applies formula (8.3) bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array $C[1..n]$ of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up
$F[0] \leftarrow 0$; $F[1] \leftarrow C[1]$
for $i \leftarrow 2$ **to** n **do**
    $F[i] \leftarrow \max(C[i] + F[i-2], F[i-1])$
return $F[n]$

$F[0] = 0, F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

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

$F[4] = \max\{10 + 5, 7\} = 15$

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

$F[5] = \max\{6 + 7, 15\} = 15$

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

$F[6] = \max\{2 + 15, 15\} = 17$

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

FIGURE 8.1 Solving the coin-row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2.

COMPLEXITY	**TIME**	**θ (n)**
	SPACE	**θ (n)**

CHANGE-MAKING PROBLEM

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0, \qquad \textbf{(8.4)}$$
$$F(0) = 0.$$

ALGORITHM *ChangeMaking*$(D[1..m], n)$
//Applies dynamic programming to find the minimum number of coins
//of denominations $d_1 < d_2 < \cdots < d_m$ where $d_1 = 1$ that add up to a
//given amount n
//Input: Positive integer n and array $D[1..m]$ of increasing positive
// integers indicating the coin denominations where $D[1] = 1$
//Output: The minimum number of coins that add up to n
$F[0] \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  $temp \leftarrow \infty$; $j \leftarrow 1$
  while $j \leq m$ **and** $i \geq D[j]$ **do**
    $temp \leftarrow \min(F[i - D[j]], temp)$
    $j \leftarrow j + 1$
  $F[i] \leftarrow temp + 1$
return $F[n]$

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1 - 1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2 - 1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3 - 1], F[3 - 3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4 - 1], F[4 - 3], F[4 - 4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5 - 1], F[5 - 3], F[5 - 4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6 - 1], F[6 - 3], F[6 - 4]\} + 1 = 2$

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

FIGURE 8.2 Application of Algorithm *MinCoinChange* to amount $n = 6$ and coin denominations 1, 3, and 4.

The Knapsack Problem and Memory Functions

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases} \qquad \textbf{(8.6)}$$

nvenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0. \qquad \textbf{(8.7)}$$

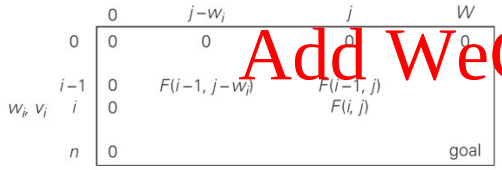


FIGURE 8.4 Table for solving the knapsack problem by dynamic programming.

		capacity j					
	i	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**

FIGURE 8.5 Example of solving an instance of the knapsack problem by the programming algorithm.

EXAMPLE 1 Let us consider the instance given by the following data:

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

capacity $W = 5$.

		capacity j					
	i	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	—	12	22	—	22
$w_3 = 3, v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2, v_4 = 15$	4	0	—	—	—	—	**37**

FIGURE 8.6 Example of solving an instance of the knapsack problem by the memory function algorithm.

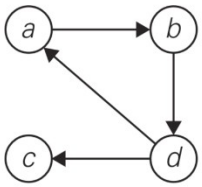
COMPLEXITY **TIME** **θ (nW)** **COMPOSITION** **θ (n)**
SPACE **θ (nW)**

Warshall's and Floyd's Algorithms

Warshall's Directed graphs

For computing the transitive closure of a directed graph

A matrix containing the information about the existence of directed paths of arbitrary lengths between vertices of a given graph is called the **transitive closure of the digraph.**



(a)

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b)

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

(c)

FIGURE 8.11 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

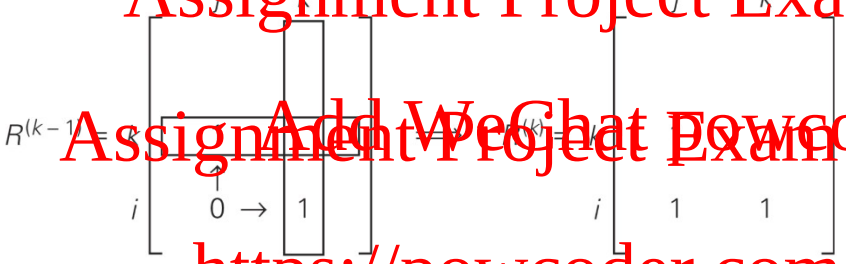
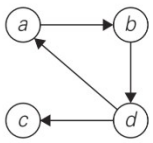


FIGURE 8.12 Rule for changing zeros in Warshall's algorithm.



$R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a, b,* and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)}$ =

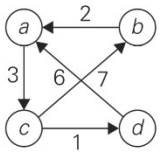
	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a, b, c,* and *d* (note five new paths).

COMPLEXITY **θ (n^3)**

Floyd's Directed and undirected graphs

all-pairs shortest-paths problem



(a)

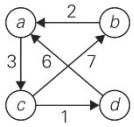
$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

(b)

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(c)

FIGURE 8.14 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.



$$D^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$$D^{(2)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note a new shortest path from *c* to *a*).

$$D^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

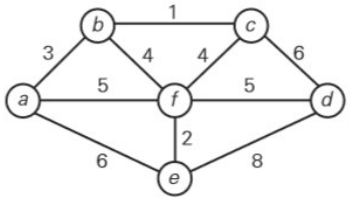
$$D^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

FIGURE 8.16 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

GREEDY ALGORTHIMS

PRIMM'S



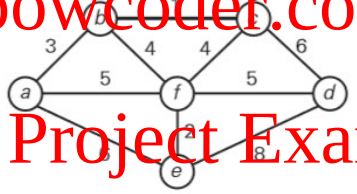
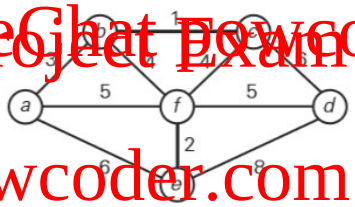
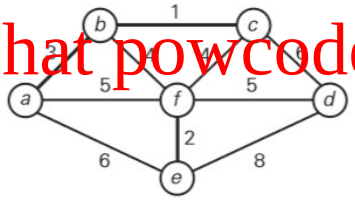
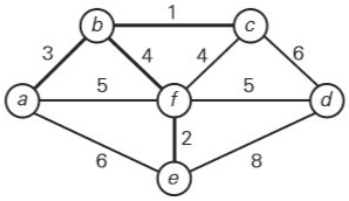
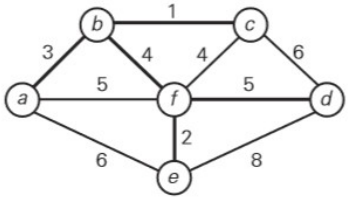
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df**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

FIGURE 9.5 Application of Kruskal's algorithm. Selected edges are shown in bold.

HUFFMAN

EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

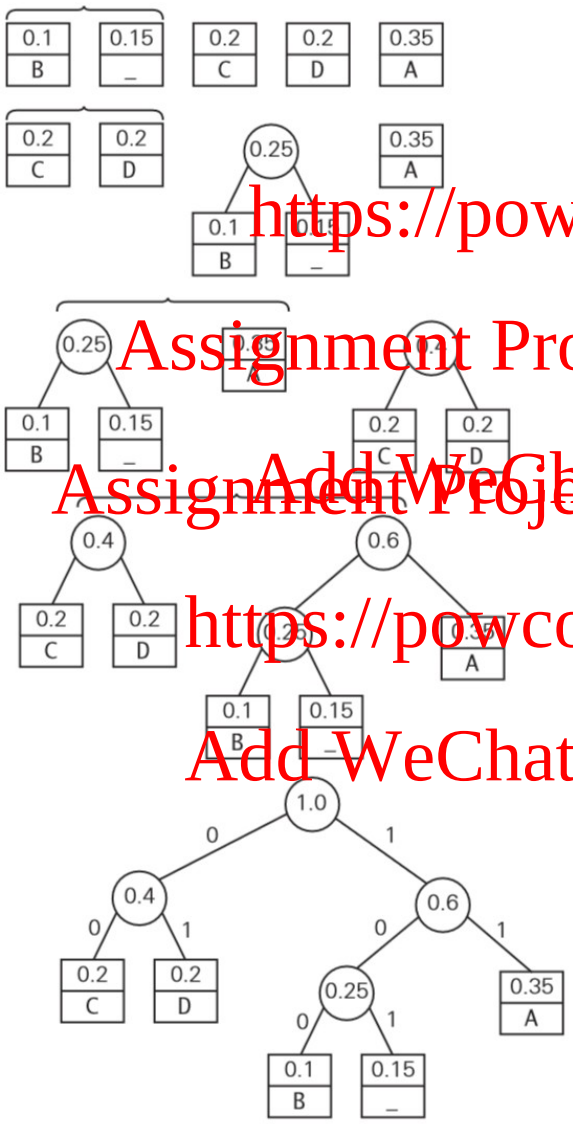


FIGURE 9.12 Example of constructing a Huffman coding tree.

The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101