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

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

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \le \cdots \le A_{n-1}$$
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**

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.

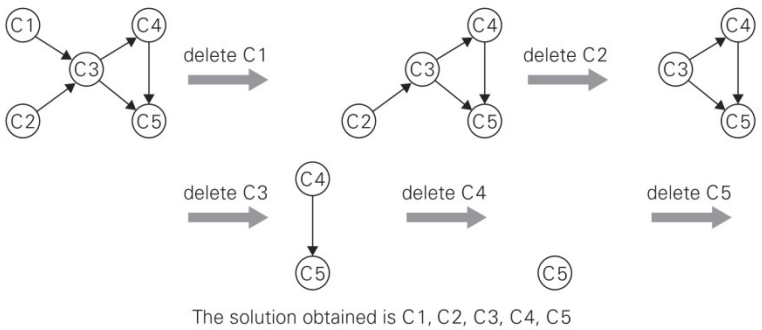


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

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

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

Based on INSERTION SORT

- Applies **insertion sort** to each of several **interleavin** . size > 10,000 or so)
- On each pass through the list, the sublists in questio ng through the list with an increment h_i taken from some predefined decre sizes, $h_1 > ... > h_i > ... > 1$, 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

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

STRATEGY: **DIVIDE AND CONQUER**

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

DISADVANTAGE over MERGESORT

COMPLEXITY:

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

BEST CASE: **$\Theta(n \log_2 n)$ 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 ← 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**

DISADVAN

COMPLEXITY AVERAGE AND WORS **Θ (n log**

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

SORTING BY COUNTING

COMPARISON SORTING COUNT

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

STABLE: **YES** **O 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 $(l \le u)$
//Output: Array $S[0..n-1]$ of A's elements sorted in nondecreasing order
for $j \leftarrow 0$ **to** $u - l$ **do** $D[j] \leftarrow 0$ //initialize frequencies
for $i \leftarrow 0$ **to** $n - 1$ **do** $D[A[i] - l] \leftarrow D[A[i] - l] + 1$ //compute frequencies
for $j \leftarrow 1$ **to** $u - l$ **do** $D[j] \leftarrow D[j - 1] + D[j]$ //reuse for distribution
for $i \leftarrow n - 1$ **downto** 0 **do**
  $j \leftarrow A[i] - l$
  $S[D[j] - 1] \leftarrow A[i]$
  $D[j] \leftarrow D[j] - 1$
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

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

BINARY SEARCH

MUST **aller files**


```
function BinSearch(A[], lo, hi, key)
    if lo > hi then return −1
    mid ← 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)
```

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 (v
AVERAGE CASE Θ (log n) precisely
WORST CASE Θ (n) (bad performance)

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 $i \leftarrow 0$ **to** $n - m$ **do**
  $j \leftarrow 0$
  while $j < m$ **and** $P[j] = T[i + j]$ **do**
    $j \leftarrow j + 1$
  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

$s_0 \quad \ldots \quad c \quad \ldots \quad s_{n-1}$

B A R B E R

FOUR CASES

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

Some Useful Formulas

From Stirling's formula:

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

Some useful sums:

$$\sum_{i=0}^{n} i^2 = \tfrac{n}{3}(n+\tfrac{1}{2})(n+1)$$

$$\sum_{i=0}^{n}(2i+1) = (n+1)^2$$

$$\sum_{i=1}^{n} 1/i = O(\log n)$$

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

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

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

EXPONE

```
function FIB(n, a, b)
    if n = 0 then
        return a
    return FIB(n − 1, a + b, a)
```

Initial call: $\text{FIB}(n, 1, 0)$

```
function FIB(n)
    a ← 1
    b ← 0
    while n > 0 do
        a ← a + b
        b ← a − b
        n ← n − 1
    return a
```

LINEAR

GRAPHS

Graph Representation

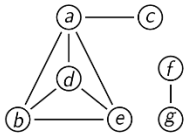
Adjacency matrix and Adjacen

(1) The **adjacency matrix** of a directed graph does not have
(2) An edge in a directed graph has just one (not two) corresp raph's **adjacency lists**.

GRAPH TRAVERSAL

DFS Stack discipline **BACKTRACKING**

Depth-First Search: The Traversal Stack



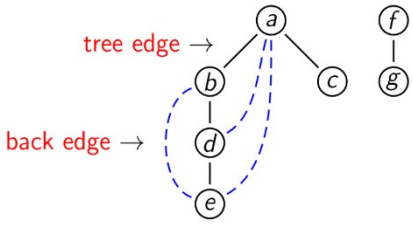
Levitin uses a more compact notation for the stack's history. Here is how the stack develops, in Levitin's notation:

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

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

BFS Queue discipline **CONCENTRIC**

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 i **smallest instance** of the problem; it is called sometimes the

- 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 ongoing edges)**
  - **ALGORITHMS FOR COMBINATORIAL OBJECTS**

- Decrease by a consta Complexity: **O (log n)**

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

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 $50 \cdot 65$ 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)

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

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

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 $T = \varnothing$ **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 (leave)**

2n + 1 = x + n

be made as Stack

IN ORDER TRAVE **R**

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

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***.

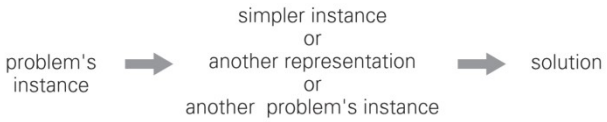


FIGURE 6.1 Transform-and-conquer strategy.

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 $O(n^2)$

COMPLEXITY Θ (n log n)
Based in SORTING time

- **COMPUTING A MODE**

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 CHAN

Create a 2-3 trees

AVL TREES

is a binary search tree but balanced

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

est (lowest) node inserted

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

log n

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

2-3 TREES

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 en -bound applications.

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:

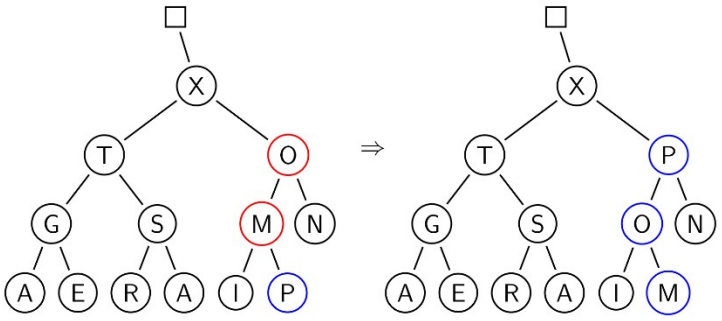
Each child has a priority (key) which is no greater than its parent's.

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.)

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)

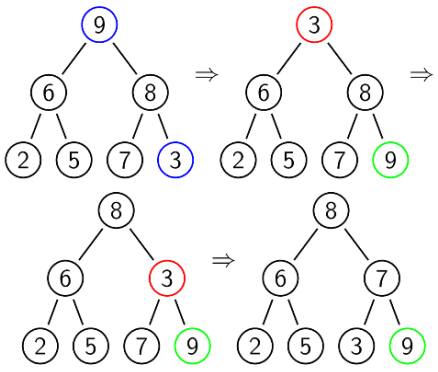
CREATION BY BOTTOM UP: **Complexity O(n)** **fewer than 2n**

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)**

TIME/SPACE TRADEOFFS

Graph traversals complexity depends on representation: n^2 (adj (adjacency list) (n = vertices, m = edges), **one better approach** is by working in as s than vertices), the use of adjacency list is powered (m ∈ O (n)).

TECHNICS

Input enhancement

Preprocess the problem's in al information obtained to accelerate solving the problem afterward.

- **Sorting by counting**
- **Boyer-Moore** algorithm for string matching and its sted 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

PROBLEMS WITH COLLISSION

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

APPROPIATELY CHOSEN **H**

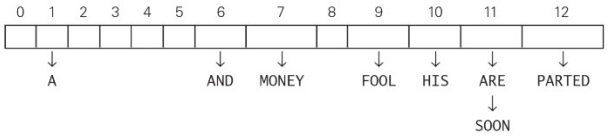
COLLISION RESOLUTION M

- **Open hashing** (sepa
- **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

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

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 P**

Assignment Project
Hash
mpty cell

Searching: Unsu
Succe ti an **empty cell**

Deleting is more complicated

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 de		
	Best best	**θ (log n)**
	But in ave	and worst cases
Ordering Preservation	**NO**	**YES**

DYNAMIC PROGRAMMING

COIN ROW PROBLEM

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

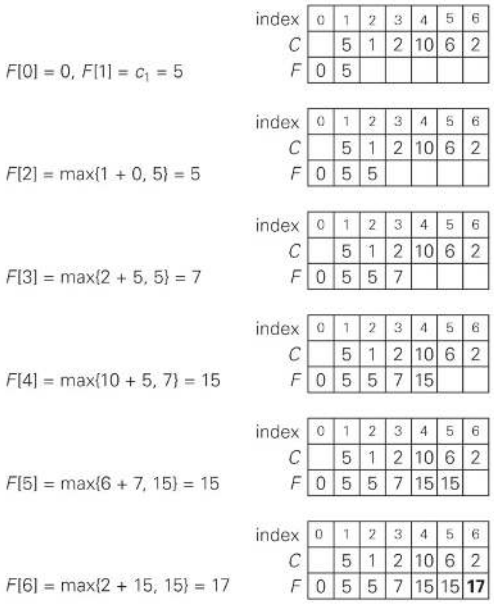


FIGURE 8.1 Solving the coin-row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2.

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

CHANGE-MAKING PROBLEM

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

The Knapsack Problem and Memory Functions

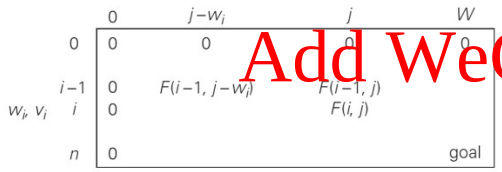


FIGURE 8.4 Table for solving the knapsack problem by dynamic programming.

		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)**
SPACE **θ (nW)**

COMPOSITION **θ (n)**

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.**

COMPLEXITY $\theta(n^3)$

Floyd's Directed and undirected graphs

all-pairs shortest-paths problem

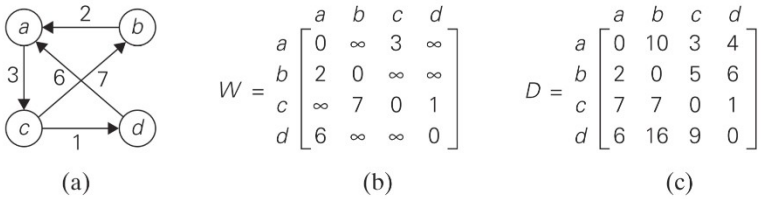


FIGURE 8.14 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

GREEDY ALGORTHIMS
PRIMM'S

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