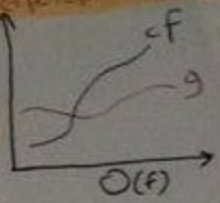
Αλγοριθμικές Συνήθειες - Ρυθμός αναπτυξης

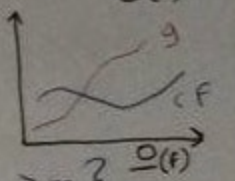
$f: N \to N$ τότε ασυμπτωτικό άνω όριο (η καλύτερη χειρότερη λύση Ο)

$O(f) = \{g: N \to N \mid \exists\, c>0, m: g(n) \le cf(n),\ \forall n \ge m\}$



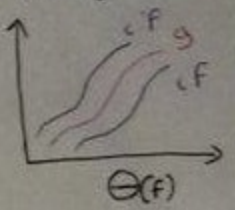
ασυμπτωτικό κάτω όριο (η χειρότερη καλύτερη λύση Ω)

$\Omega(f) = \{g: N \to N \mid \exists\, c>0, m: g(n) \ge cf(n),\ \forall n \ge m\}$



Ασυμπτωτικό σφιχτό όριο

$\Theta(f) = \{g: N \to N \mid \exists\, c, c' > 0, m: cf(n) \le g(n) \le c'f(n),\ \forall n \ge m\}$



$o(f)$: $g(n) < cf(n)$ } strict grater/smaller
$\omega(f)$: $g(n) > cf(n)$ }

Σύγκριση συναρτήσεων

$o(1) \subset O(1) \subset O(\log n) \subset O(x \log n) \subset O(n^e)$

$\subset O(n^x) \subset O(n^x \times \log n) \subset O(2^n) \subset O(n!) \subset O(n^n)$

Διαισθητικά για αλγορίθμους:

$O(1)$: Όταν ο αλγόριθμος δεν έχει loop ή έχει και γνωρίζουμε ότι κάποια συγκεκριμένη φορά θα σταματήσει. (πχ for (i=1; i<5; i++))

$O(n)$: Όταν ο αλγόριθμος έχει 1 loop (μέχρι n)

$O(\log n)$: Όταν ο αλγόριθμος έχει 1 loop, το οποίο όπως λέμε ότι "χάνεται" στη μέση.

$O(n^2)$: Όταν ο αλγόριθμος έχει 2 loop το ένα μέσα στο άλλο

Example operations number

(Input array X of n integers
Output array A of prefix averages of X)

	operations
A ← new array of n integers	n
s ← 0	1
for i ← 0 to n-1 do	n
s ← s + X[i]	n
A[i] ← s/(i+1)	n
return A;	1

$\Longrightarrow O(n)$

④ Cuckoo Hashing

Έχουμε 2 hash functions $h1()$ και $h2()$ για το κάθε στοιχείο που θέλουμε να εισάγουμε. Δίνοντας 2 θέσεις στο hash table. Αν η πρώτη είναι άδεια, βάζουμε εκεί το στοιχείο. Αν η πρώτη είναι γεμάτη και η δεύτερη άδεια, βάζουμε στην δεύτερη το στοιχείο. Αν και οι 2 θέσεις είναι γεμάτες, βάζουμε το στοιχείο στην πρώτη, και βγάζουμε αυτό που ήταν εκεί. Τώρα ξανα περνάμε από τις hash functions το στοιχείο που βγάλαμε βάζοντας το στην άλλη θέση από αυτή που ήταν. Αν στην άλλη εκείνη θέση υπάρχει άλλο στοιχείο, το βγάλαμε και επαναλαμβάνουμε την διαδικασία για εκείνο.

Αν infinit loop → rebuild hash table.

```
procedure Insert(x)
    if T[h1(x)] = x or T[h2(x)] = x then return;
    pos ← h1(x);
    loop n times {
        if T[pos] = NULL then {T[pos] ← x; return};
        x ↔ T[pos];
        if pos = h1(x) then pos ← h2(x) else pos ← h1(x);}
    rehash(); insert(x)
end
```

Lookup $O(1)$

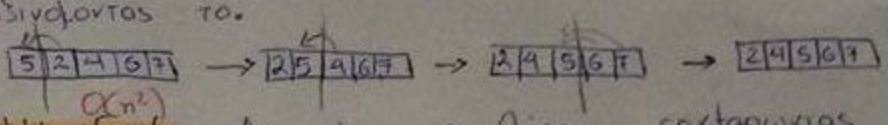
rehashing $O(1/n)$

Πολυπλοκότητες γνωστών Functions

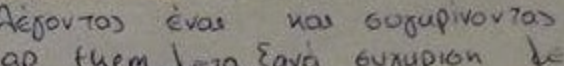
	best	worst	worst space	
Bubble Sort	$O(n^2)$	$O(n^2)$	$n + O(1)$	good for small inputs
Insertion Sort	$O(n^2)$	$O(n^2)$	$n + O(1)$	
Selection Sort	$O(n^2)$	$O(n^2)$	$n + O(1)$	
Heap Sort	$O(n \log n)$	$O(n \log n)$	$n + O(1)$	good for large inputs
Merge Sort	$O(n \log n)$	$O(n \log n)$	$2n + O(1)$	good for huge inputs
Quick Sort	$O(n \log n)$	$O(n^2)$		good for large inputs
Binary Search	$O(1)$	$O(\log n)$	$O(1)$	

Αλγόριθμοι ταξινόμησης

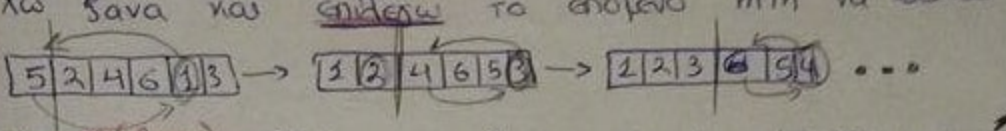
Insertion Sort $O(n^2)$: Θεωρώ πρώτο στοιχείο sorted και μετά επιλέγω (Εισάγω) το αμέσως επόμενο στο sorted κομμάτι της λίστας ταξινομώντας το.



Bubble Sort $O(n^2)$: Διατρέχω τη λίστα, sortαρωντας 2άδες στοιχείων, επιλέγοντας έναν και συγκρίνοντας του με το δεξιανό του και swap them, μετά ξανά συγκριση με το επόμενο κλπ. Διατρέχω όσες φορές χρειαστεί.



Selection Sort $O(n^2)$: Βρίσκω το min, το βάζω πρώτο και μετά διαρκώς ξανά και ξανά επιλέγω το επόμενο min να βάλω 2ο. (swap)

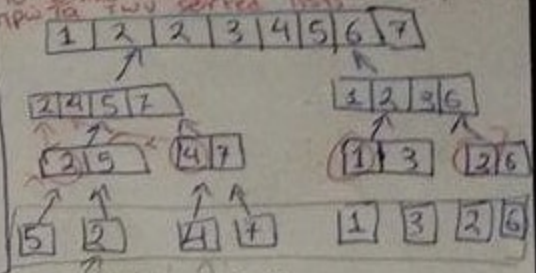


$$T(n)=\begin{cases} b & n>1 \\ 2T(n/2)+bn \end{cases}$$

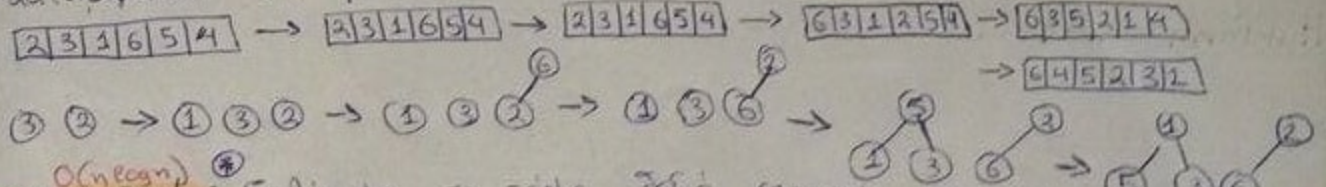
Merge Sort $O(n\log n)$: Σπάμε την λίστα σε μικρότερες μέχρι να έχουμε μονά στοιχεία. Μετά αρχίζουμε να την ξανα χτίζουμε ενώνοντας και sortαροντας 1-1, 2-2, 3-3 ... τα στοιχεία-υποσύνολα.
Divide & conquer

με :

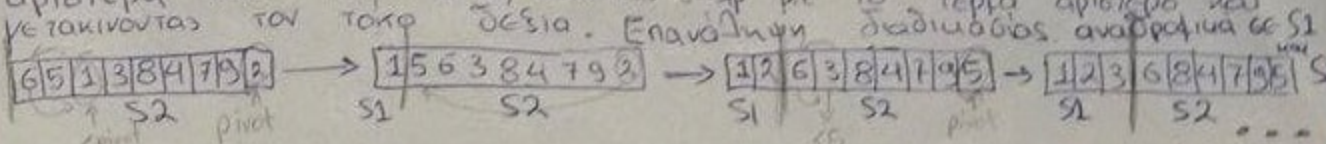
```
void MergeSort(table A, int p, r) {
        IF (p<r){
            q=L(p+2)/2];
            MergeSort(A,P,q);
            MergeSort(A,q+1,r);
            Merge(A,P,q,r);
        }
}
```



Heap Sort $O(n\log n)$ (*): Λίστα μερικώς ταξινομημένη σωρός, το μη ταξινομημένο τμήμα της λίστας. Επαναληπτική αποθήκευση του μικρότερου στοιχείου του σωρού στη θέση i του πίνακα. Επαναφορά της κεντρικής διάταξης του σωρού που μπορεί να έχει καταστραφεί στη ρίζα.
Divide & conquer



Quick Sort $O(n\log n)$ (*): Επιλέγουμε το τέρμα δεξιά στοιχείο ως pivot.
Divide & conquer Μετά βάζουμε ένα τοίχο τέρμα αριστερά και συγκρίνουμε τα επόμενα στοιχεία 1-1 με το pivot. Αν είναι < pivot τα βάζουμε αριστερά από τον τοίχο, κάνοντας swap με το τέρμα αριστερό και μετακινούμε τον τοίχο δεξιά. Επαναληψη διαδικασίας αναδρομικά σε S1 και S2.



Best case: Αν το pivot είναι στη μέση των συνολικών στοιχείων $O(n\log n)$ / Worst: $O(n^2)$

Lower Bound

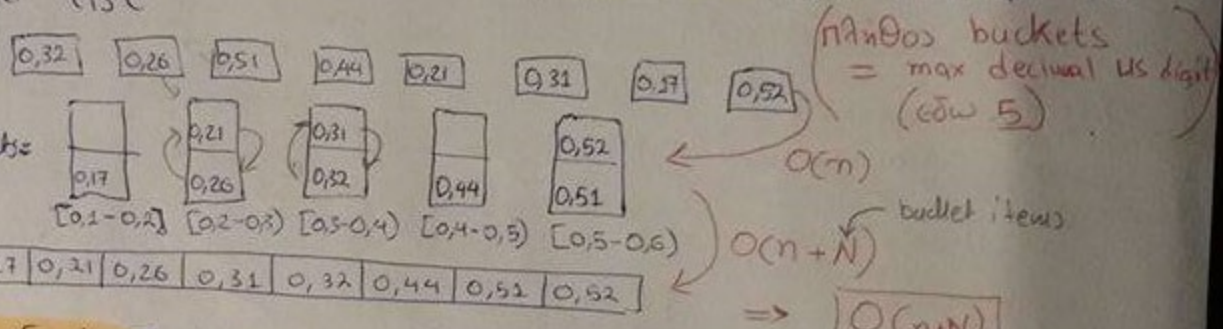
Κάθε αλγόριθμος βασισμένος σε σύγκριση, θέλει τουλάχιστον $\log(n!)$ χρόνο.

Οπότε κάθε τέτοιος αλγόριθμος θέλει χρόνο τουλ:

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log(n/2)$$

Γι' αυτό κάθε comparison-based sorting algorithm πρέπει να τρέχει σε $\Omega(n \log n)$

Bucket Sort: Επιλέγουμε n buckets, όπου n = το πλήθος των διαφορετικών στοιχείων της λίστας, βάζουμε τα στοιχεία στα buckets (stack) και τα ταξινομούμε αν χρειάζεται σε κάθε bucket και με pop από το χαμηλότερο bucket χτίζουμε την sorted list



Radix Sort: Sortaroume την λίστα κατά τα least significant digits και μετά για τα επόμενα σημαντικά digits κλπ, κλπ

Worst $O(k \cdot n)$

[359, 383, 598, 911, 479, 544]

Δεν είναι comparison-based!

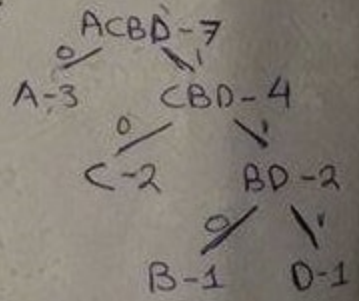
#1	#2	#3
911	911	359
383	544	383
594	548	479
548	359	544
359	479	548
479	383	911

Huffman Algorithm (greedy)

Μετατροπή data σε bits με το χαμηλότερο κόστος.

Αλγόριθμος: Μετράω συχνότητες εμφάνισης. Τις sortάρω. Πέρνω τα 2 χαμηλότερα και τα βάζω σε 2 φύλλα δένδρου με πατέρα το αθροισμά τους. Βάζω τον πατέρα στην λίστα και βγάζω τα 2 παιδιά. Για αναπαράσταση σε bits από ρίζα, αν LC ⇒ 0 Αν RC ⇒ 1

Π.Χ	ABACCDA
A-3	
C-2	
B-1	
D-1	



A = 0, B = 110 D = 111 C = 10

Greedy Method - Knapsack

Knapsack χωράει 20 kg. Αντικείμενα με διαφορετική αξία και βάρος.
Αν πρέπει να πάρουμε ένα ακέραιο αντικείμενο = 0/1 Knapsack
Αν μπορούμε να πάρουμε μερικό κομμάτι ενός αντικειμένου = Fractional Knapsack

0/1 Knapsack (με dynamic programming)

Ενας greedy algorithm δεν μπορούμε να δείξουμε αν αποδώσει. Κάθε φορά εξαρτάται από τα βάρη και την αξία ο βέλτιστος συνδιασμός.

Fractional Knapsack (με greedy)

Εδώ ένας greedy algorithm είναι η βέλτιστη λύση, (για Fixed weight W).

Π.χ

νερό	περπέλαιο	κρασί
10$	100$	30$

Πρώτα βάζω όσο περπέλαιο μπορώ μετά κρασί, μετά νερό

Master Theorem

Recurrence time of Merge Sort:

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

Base → overhead of merging (linear)

Master theorem: Για κάθε αναδρομικού χρόνου συναρτήσεων τύπου:

General

$$T(n) = aT(n/b) + f(n)$$
$$T(1) = c$$

με $a \geq 1$, $b \geq 2$, $c > 0$

και αν $f(n) = O(n^d)$ όπου $d \geq 0$, τότε

το $T(n)$ μπορεί να πάρει τις τιμές:

$$T(n) = \begin{cases} O(n^d) & \text{αν } a < b^d \\ O(n^d \log n) & \text{αν } a = b^d \\ O(n^{\log_b a}) & \text{αν } a > b^d \end{cases}$$

Matrix chain multiplication Cost

Π.χ

0	1	2	3
[2,3]	[3,6]	[6,4]	[4,5]

① $(0\cdot1)(2\cdot3) = 2\cdot3\cdot6 + 6\cdot4\cdot5 + 2\cdot6\cdot5 = 36 + 120 + 60 = 216$

② $((0\cdot1)2)3 = 2\cdot3\cdot6 + 2\cdot6\cdot4 + 2\cdot4\cdot5 = 36 + 48 + 40 = 124$

	0	1	2	3
0	0	36 (0,1)	(0,1)(2,2) F:1	(0,2)(3,3) 124
1		0	(1,2) 2,2	(1,2)(3,3) 132
2			0	(2,3) 120
3				0

min cost!

Για τη διαγώνιο (0,2)-(1,3) [e=3]

Θέλω να πολ/σω 0·1·2 → 0·(1·2) ①, 2·(0·1) ②
[2,3] [3,6] [6,4]

① = 72 + 2·3·4 = 96

② = 36 + 2·6·4 = 84 better

Knapsack 0/1, Dynamic programming example

Χώρος Knapsack = 5 Kg

Weights : 2 Kg 3 Kg 4 Kg 5 Kg

Benefits : 3 7 2 9

items \ weight		0 Kg	1 Kg	2 Kg	3 Kg	4 Kg	5 Kg
	0	0	0	0	0	0	0
2 Kg	1	0	0	3	3	3	3
2,3 Kg	2	0	0	3	7	7	10
2,3,4 Kg	3	0	0	3	7	7	10
2,3,4,5 Kg	4	0	0	3	7	7	(10)

2 items αυτή τη φορά

5kg = 2kg + 3kg = 3 + 7 = 10

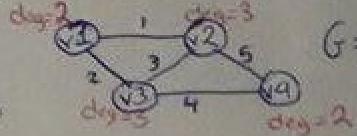
(10) → Best

Graphs

Δομή G = (V, E) V = πόσες κορυφές Vertices (κόμβοι)

E = σύνολο των δεσμών των Vertices.

Degree of vertex : το πλήθος των edges που συνδέονται με τον κόμβο



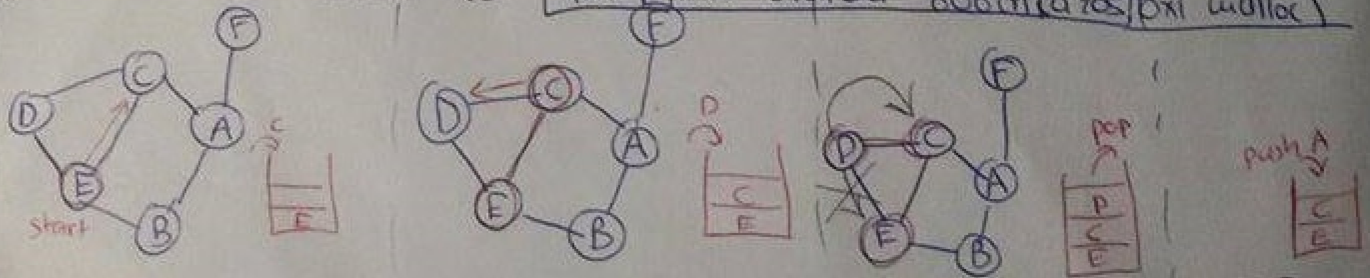
G = (4, 5)

Adjacency Matrix

	v1	v2	v3	v4
v1	0	1	1	0
v2	1	0	1	1
v3	1	1	0	1
v4	0	1	1	0

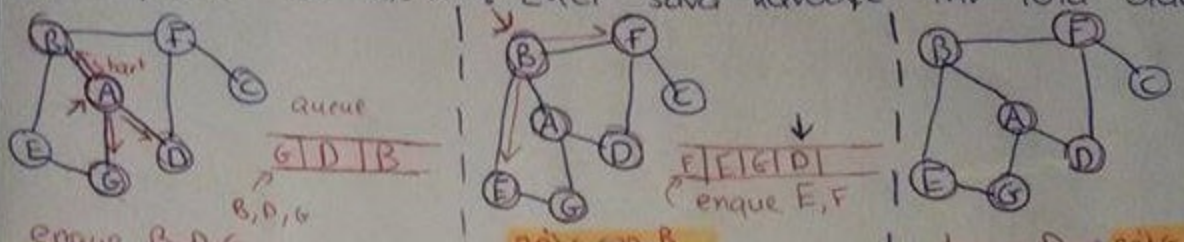
DFS Depth-First-Search (brute-force) O(V+E)

Ξεκινάμε απο ένα δοσμένο κόμβο στον γράφο και κινούμαστε οποιοδήποτε ανεξερεύνητο κόμβο. Κάθε κόμβο που συναντάμε τον βάζουμε σε ένα βοηθητικό stack. Αν φτάσουμε σε κόμβο που πάει μόνο σε εξερευνημένους, πάμε προς τα πίσω μέχρι να φτάσουμε σε κόμβο που έχει επιλογές (κάνοντας pop απο την στοίβα). Αναδρομική στοίβα συστήματος όχι μόνο



BFS Breadth-First Search O(V+E) (sortest paths)

Ξεκινάτε την διάσχιση από οποιονδήποτε κόμβο ενός γράφου. Χρησιμοποιούμε μια ουρά (queue) σαν βοηθητική δομή. Από τον κόμβο που βρισκόμαστε κάνουμε enque και χαρακτηρίζουμε ως visited όλους τους συνδεδεμένους με αυτόν. Τότε κάνουμε deque από την ουρά (τον πρώτο που βάλαμε) και θεωρούμε τώρα ως βρισκόμαστε σε αυτόν! Εκεί ξανά κάνουμε την ίδια διαδικασία.

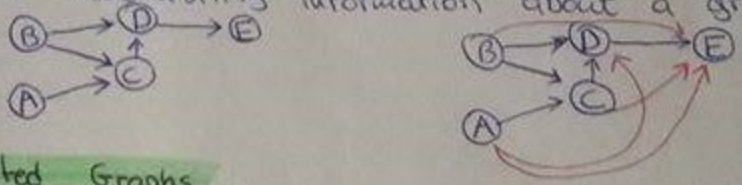


Applications	DFS	BFS
Spanning forest, paths, cycles, cohected components	✓	✓
sortest paths		✓
Biconnected compa	✓	

Trausitive closure on digraph

provide reachability information about a graph.

π.χ



Weighted Graphs

Γράφοι των οποίων τα edges έχουν βάρη που καθορίζουν το κόστος της διαδρομής από τον κόμβο 1 στον 2.

Floyd-Warshall algorithm (shortest paths) - Weighted Graphs

Αλγόριθμος για κάθε ζευγάρι κορυφών, το συντομότερο μονοπάτι μεταξύ τους.

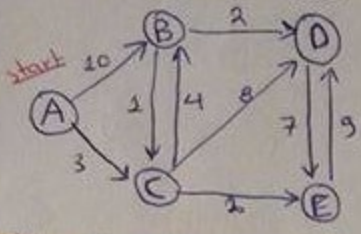
```
Για κάθε vertex v
    dist[v][v] ← 0
Για κάθε edge (u,v)
    dist[u][v] ← w(u,v)
Για k από 1 σε |V|
    Για i από 1 σε |V|
        Για j από 1 σε |V|
            Αν (dist[i][j] > dist[i][k] + dist[k][j]) {
                dist[i][j] ← dist[i][k] + dist[k][j]
```

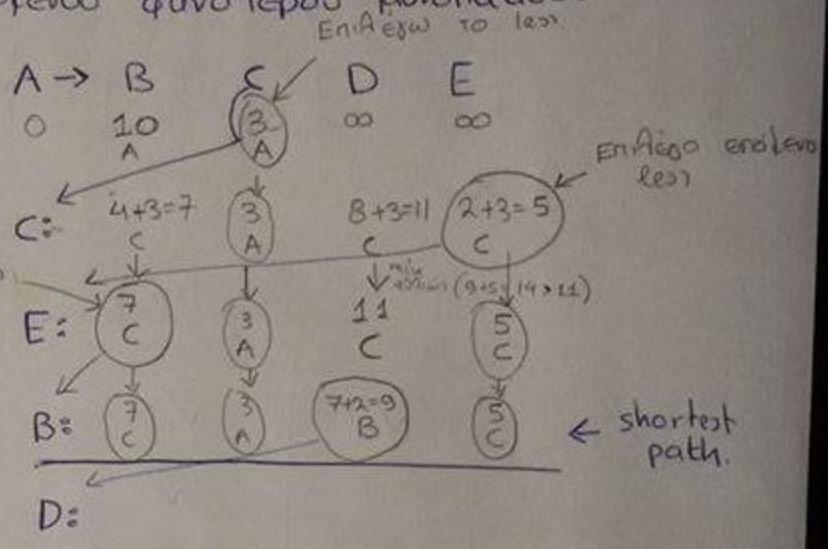
Dijkstra's Algorithm

$O(V \log V + E \log V) = O(E \log V)$ το

Βασιζόμενος στην λογική του ότι το σύνολο των πιο μπήγορων μονοπατιών μεταξύ 2 κόμβων κάθε φορά είναι εν τέλει το πιο σύντομο μονοπάτι.

Από εκεί που ξεκινάμε, ακολουθούμε το φτηνότερο μονοπάτι και μετά από εκεί που μας πάει το φτηνότερο αυτό επιλέγουμε το εποπτευο φτηνότερο κλπ... Προσθέτοντας κάθε φορά το κόστος του προηγούμενου φτηνότερου μονοπατιού.





Αλγόριθμος Dijkstra: next less

Αρχικοποιούμε την αρχική κορυφή με 0 και τις άλλες με ∞. Εισάγουμε σε μία ουρά όλες τις κορυφές. και για μία μία κάνουμε deque το shortest path μέχρι να αδειάσει η ουρά

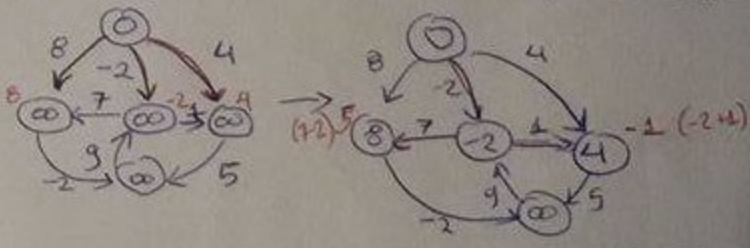
↳ Δεν δουλεύει για negative weight edges!

Bellman-Ford Algorithm

→ Δουλεύει και για negative weight edges

$O(n \cdot m)$ Ίδια λογική με του Dijkstra, + προσθέτουμε σε κάθε κόμβο το $d(v)$ value του:

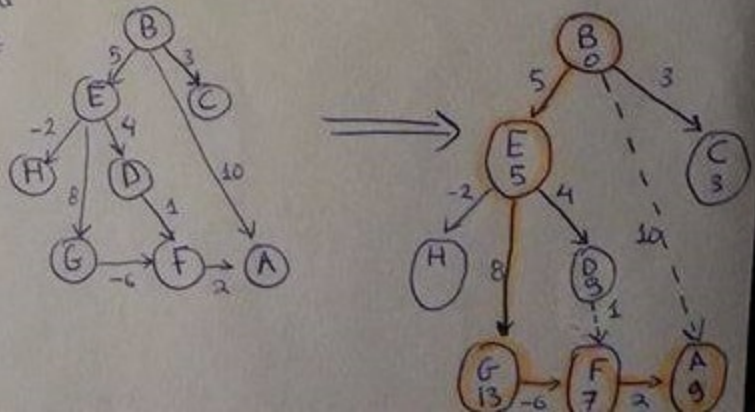
Ξανά κάνει διαδρομή αν χαλάσουν τα $d(v)$.
(Αν δεν συναντήσει αρνητικό weight είναι ίδιος με Dijkstra)



...

Directed Acyclic Graph Shortest Paths - DAG -

Ξεκινάμε με 0 όπως πάντα και τα υπόλοιπα ∞, και προχωράμε επίπεδα. (easy) Κάνοντας relax κάθε edge



Minimum Spanning Tree – MST –

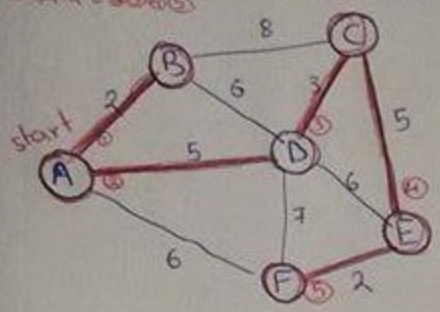
Spanning tree: ένας υπογράφος που αποτελεί κόμο του ένα δένδρο με όλα τα vertices.

Minimum spanning tree: Ένα spanning tree ενός weighted graph με το ελάχιστο συνολικό βάρος edge.

Prim's Algorithm $O(E \log V)$

Για να ξεκινήσεις minimum spanning trees σε graphs.

Βήματα



step 1: Επιλέγουμε οποιοδήποτε starting point. Επιλέγουμε τον κόμβο που ενώνεται με αυτόν με το min λιγότερο βάρος (εδώ το Β)

step 2: Τώρα κοιτάμε όλα τα edges που συνδέονται στο δένδρο μας και επιλέγουμε αυτό με το μικρότερο βάρος.

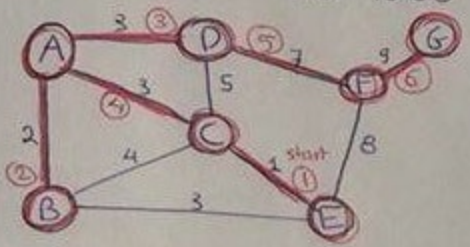
step 3: repeat step 2. μέχρι να πάρουμε όλα τα vertices.

Kruskal's Algorithm

Χρόνος εξαρτάται από τον χρόνο sortarismatos των edges. Με merge-sort → $O(E \log E)$

Επιλέγουμε το edge με το μικρότερο βάρος, και τους κόμβους που ενώνει. Μετά συνεχίζουμε με τα αμέσως επόμενα σε βάρος edges κλπ... (επιλέγοντας κατά προσοχήν να μην ενώσουμε 2 vertices του ίδιου δένδρου ξανά μεταξύ τους)

π.χ.



<Edges σε ουρά προτεραιότητας>

$O(E \log V)$ Greedy

Σε μικρό γράφο προτιμώ Kruskal για να παίζω με κόμβους και όχι edges!

Έχω sorted όλα τα edges σε ουρά προτεραιότητας και το περνώ 1-1. Σε αντίθεση με τον Prim που κάθε φορά πέρνω edge που να συνδέεται στο μέχρι στιγμής δένδρο.

Ford Fulkerson

Για να βρίσκουμε το maximum flow σε ένα flow network από ένα source (s) σε ένα sink (t).

Κάποιοι όροι για Flows in Networks

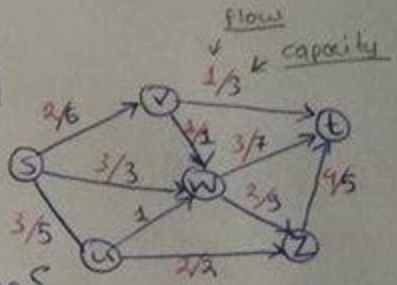
1) Για κάθε vertex $u, v \in V$ το flow από u στο v πρέπει να είναι ≥ 0 και $\leq$ της capacity του edge μεταξύ u και v

$\forall u,v \in V \Rightarrow 0 \leq f(u,v) \leq c(u,v)$

2) Για όλες τις vertex εκτός από u, v εκτός από s και t το συνολικό flow από το u στο v πρέπει να είναι ίσο με το συνολικό flow από το v στο u

$\forall u \in V - \{s,t\} \Rightarrow \sum_{v \in V} f(u,v) = \sum_{v \in V} f(v,u)$

Π.χ

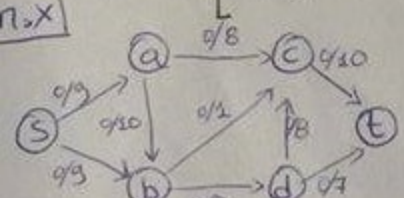


Αλγόριθμος: ενώνει s-t

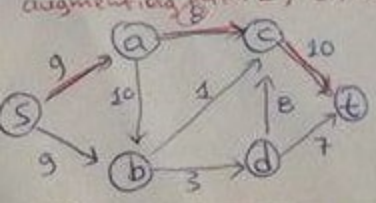
```
Max Flow = 0
Όσο υπάρχει augmenting path στο γράφο {
    Βρες το augmenting path;
    Βρες την ελάχιστη capacity του path;
    Πρόσθεσε την στο Max Flow;
    Για κάθε edge στο augmentin path {
        • Αν είναι original edge, μείωσε την capacity της κατά την ελάχιστη capacity.
        • Αν είναι return edge, αύξησε την capacity της κατά την ελαχίστη capacity;
    }
}
```

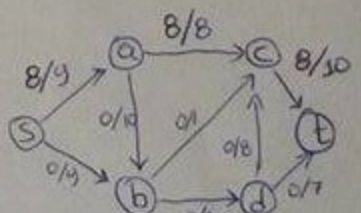
κάθε διάσχιση $O(E)$ // E: n. of edges $\Rightarrow O(E \cdot \text{maxFlow})$
θα δίνει Max Flow φορές

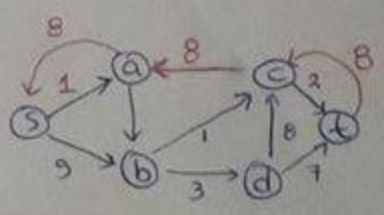
Π.χ



augmenting path 1, el capacity = 8







Επιλέγω επόμενο augmenting path
...

Επόμενο el capacity = ③ bad
και επόμενο = ① bad

$\Rightarrow$
Max Flow = 0+8+3+1
= 12

Complexity class P

P είναι μια συλλογή από γλώσσες που γίνονται δεκτές από πολυωνυμικού χρόνου αλγορίθμους.

(complexity class)

Complexity class NP

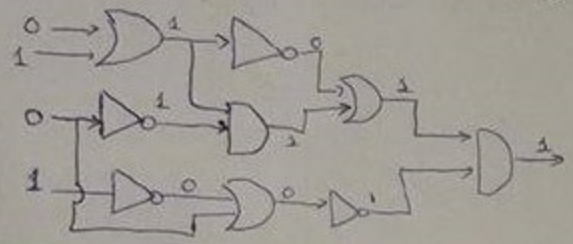
NP είναι μια συλλογή από γλώσσες (complexity class) που γίνονται δεκτές από πολυωνυμικού χρόνου μη ντετερμινιστικούς αλγορίθμους

χρησιμοποιώντας την θεωρία της επιλογής

CIRCUIT-SAT problem (NP)

Για να δεις αν υπάρχει μια δίλιση 0 και 1 στο input ενός boolean circuit ώστε το output να δίνει 1.

Μη ντετερμινιστικά επιλέγουμε ένα set από inputs και το αποτέλεσμα της κάθε πύλης.



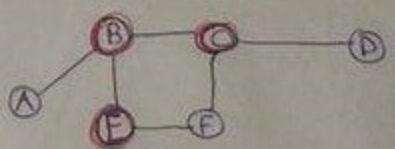
Vertex Cover (NP)

Είναι το ελάχιστο πλήθος των vertices που χρειάζονται σε ένα γράφο, ώστε να καλύπτουν όλα τα edges:

Π.χ)

B,C αρκούν

Π.χ 2)



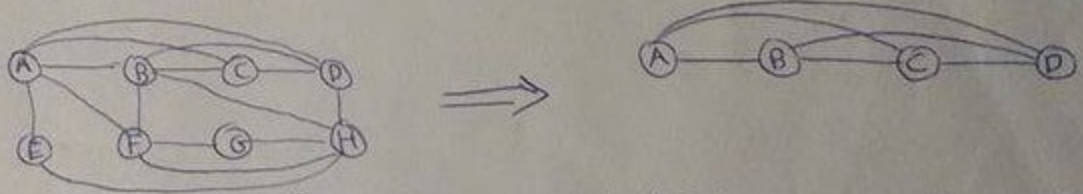
B,C,E

Μη ντετερμινιστικά επιλέγουμε ένα υποσύνολο του γράφου, και ελέγχουμε αν ένα ένα τα edges καλύπτωνται.

Clique Problem (NP)

Κάθε κόμβος (vertex) συνδέεται με όλους τους άλλους κόμβους.

Π.χ)



κοιτάμε το degree κάθε κόμβου αρχικά (βλέποντας να είναι όλοι ίδιοι)

Μη ντετερμινιστικά επιλέγουμε ένα υποσύνολο του γράφου και κοιτάμε τα degrees να είναι ίδια όλα

Hamiltonian Cycle (NP)

Αν υπάρχει δρομολόγιο σε ένα γράφο που να επισκέπτεται κάθε κορυφή (vertex) ακριβώς μία φορά.

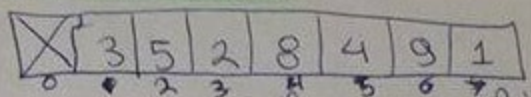
Απόδειξη NP

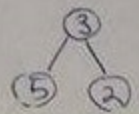
Έχω μια μη ντετερμινιστική μηχανή Turing, που διαλέγει κάποιο μονοπάτι (την λύση). Και στη συνέχεια επιβεβαιώνει ότι το μονοπάτι είναι απλό (τη συνθήκη).

Απόδειξη NP complete

Συνήθως απόδειξη ότι δεν μπορεί να λυθεί το πρόβλημα σε P. Ανάγω το πρόβλημα σε ένα πρόβλημα που έχει αποδειχθεί ότι είναι NP complete (Hamilton - CIRCUIT-SAT κλπ)

Heap Sort example

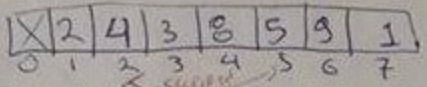


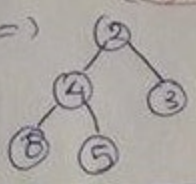


Γεμίζω το πλήρες δυαδικό δένδρο με τα στοιχεία.

κανόνας: Πατέρας να είναι πάντα μικρότερος από τα παιδιά. (μεγαλύτερος, όλη)

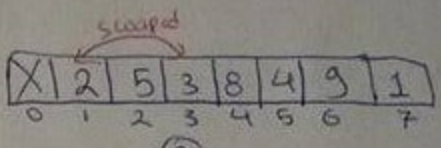
Γεμίζω τον πίνακα: πατέρας, LC, RC

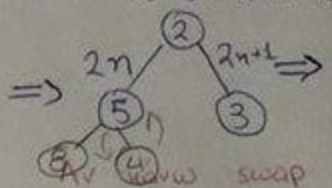




Μόλις φτάσεις το 1 στη ρίζα ως μικρότερος όλων, το αφαιρούμε από το δένδρο και ξανα εφαρμόζουμε τις ιδιότητες για ξανά πάω στη ρίζα το min.

<Κάθε κύκλο πέρνω το min των στοιχείων sortarismeno>





Αν κάνω swap 2 κόμβους

Αν ήταν δεξί φύλλο κάνω swap στην ρίζα με τη θέση 2n+1

Αν ήταν αριστερό παιδί LC το 2n

FFT - Fast Fourier Transform

Χρησιμοποιείται για τον υπολογισμό DFT (Discrete Fourier Transform) πιο γρήγορα.

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$$

Αν n είναι άρτιος μπορώ να το χωρίσω σε 2 πολυώνυμα

$$P^{\text{άρτιος}}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1}$$

$$P^{\text{περιττός}}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1}$$

$$\Rightarrow \boxed{P(x) = P^{\text{άρτιος}}(x^2) + x P^{\text{περιττός}}(x^2)}$$

Quicksort code & analysis

```
Quicksort (A, start, end){
  if (start < end) {
    pIndex ← Partition (A, start, end)
    Quicksort (A, start, pIndex-1)
    Quicksort (A, pIndex+1, end)
  }
}
```

Best case: Κάθε φορά το partition να χωρίζει τον πίνακα στη μέση.

Analysis Best case

$T(1) = c_1$

$T(n) = 2T(n/2) + c \cdot n$

$= 2\left[2T(n/4) + c \cdot \frac{n}{2}\right] + c \cdot n$

$= 4T(n/4) + 2cn$

$= 8T(n/8) + 3cn$

$= 2^k T(n/2^k) + kcn$

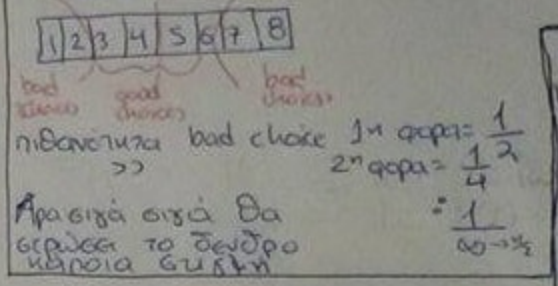
$\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$

$= 2^{\log_2 n} \cdot T(1) + c \cdot n \log_2 n$

$= n \cdot c_1 + c\, n \log n \Rightarrow O(n \log n)$

Average Case

Με πιθανότητες



πιθανότητα bad choice 1η φορά: $\frac{1}{2}$

2η φορά: $\frac{1}{4}$

>>

Άρα σιγά σιγά θα σπάσει το δένδρο κοντεύει στο $\frac{1}{\infty \to \frac{1}{2}}$

Η' πιο απλά με Master Theorem

αφού

$a = b^d$

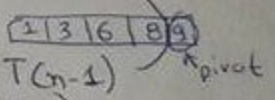
$4 = 4^1$

$\Rightarrow O(n^d \log n) = O(n \log n)$

Analysis Worst case

Worst case: Αν σε κάθε partition όλα τα στοιχεία είναι μεγαλύτερα από το pivot ή όλα μικρότερα. (το δένδρο θα αναπτυχθεί προς μία κατεύθυνση LC ή RC.)

Σε αυτή την περίπτωση π.χ.



κάθε quicksort partition θα κοστίζει $T(n-1)$ διότι θα τα διατρέξει όλα (εκτός το pivot).

$\Rightarrow$

$T(1) = c_1$

$T(n) = T(n-1) + c \cdot n$

$= T(n-2) + 2cn - c$

$= T(n-3) + 3cn - 3c$

$= T(n-4) + 4cn - 6c$

$= T(n-k) + kcn - \frac{k(k-1)}{2} \cdot c$

Έστω $T(1) = c \Rightarrow n - k = 1 \Rightarrow k = n$

$= T(1) + cn^2 - \frac{n(n-1)}{2} \cdot c = c_1 + \frac{cn(n+1)}{2} = \frac{cn^2}{2} + \frac{cn}{2} + c_1 = O(n^2)$

Master Theorem | All

$$T(n)=\begin{cases} c & \text{αν } n<d \\ aT(n/b)+f(n) & \text{αν } n\geq d \end{cases}$$

Τι ποσοστό του προβλήματος λύνει το κάθε υποπρόβλημα.

α είναι ο αριθμός των υποπροβλημάτων χωρίς το προβλήμα

n βαριά δουλειά (ή το divide ή το conquer)

Master Theorem

1. Αν $f(n)$ είναι $O(n^{\log_b a-\varepsilon})$, $\Rightarrow T(n)=\Theta(n^{\log_b a})$
2. Αν $f(n)$ είναι $\Theta(n^{\log_b a}\log^k n) \Rightarrow T(n)=\Theta(n^{\log_b a}\log^{k+1} n)$
3. Αν $F(n)$ είναι $\Omega(n^{\log_b a+\varepsilon}) \Rightarrow T(n)=\Theta(f(n))$

Υπο την προυποθεση $af(n/b)\leq \delta f(n)$ για $\delta<1$

Example 1

$T(n)=4T(n/2)+n$

$\log_b a=\log_2 4=2 \Rightarrow$ case 1.: $T(n)=O(n^2)$

Example 2

$T(n)=2T(n/2)+n\log n$

$\log_b a=\log_2 2=1 \Rightarrow$ case 2: $T(n)=O(n\log^2 n)$

Example 3

$T(n)=T(n/3)+n\log n$

$\log_b a=\log_3 1=0 \Rightarrow$ case 3 : $T(n)=O(n\log n)$

$$T(n)=\begin{cases} \Theta(n^{\log_b a}) & \text{αν } F=O(n^{\log_b a-\varepsilon}) \quad a<b^d \\ \Theta(n^{\log_b a}\log^{k+1} n) & \text{αν } F=\Theta(n^{\log_b a}\log^k n) \quad a=b^d \\ \Theta(f(n)) & \text{αν } F=\Omega(n^{\log_b a+\varepsilon}) \quad a>b^d \end{cases}$$

$af(n/b)\leq \delta f(n)$ για $\delta<1$