SFWR ENG 2C03 Summary

Author: Kemal Ahmed
Instructor: Dr. Karakostas
Date: Winter 2014
Math objects made using MathType; graphs made using Winplot.

Please join GitHub and contribute to this document. There is a guide on how to do this on my GitHub.

Table of Contents

Chapter 1

Algorithm correctness

3 things to prove:

- Initialization
- Maintenance

- Termination

An algorithm is **correct** if its output is correct. This can be measured by looking at the loop invariant.

Loop invariant: a property that will be true through all iterations; it proves correctness in algorithms; initialization, maintenance, termination; e.g. Insertion Sort is always sorted on the left side

For a given array, a,
loop invariant: a[1: j − 1] is going to be sorted

Chapter 2

Insertion Sort

Insertion sort: know the pseudo-code
e.g. stack of cards with numbers: 6, 8, 4, 7, 3
j = current card
i = previous card
key = value at current card
Never taking array index of 0

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

$T(n) = an^2 + bn + c'$

BC = 3 4 6 7 8 O(n)
WC = 8 7 6 4 3 $O(n^2)$

n = # of elements in array

Running times by line:
C1: n
C2: n – 1
C4: n − 1
C5: $\sum_{j=2}^{n} t_j$

C6: $\sum_{j=2}^{n} t_j - 1$

C7: $\sum_{j=2}^{n} t_j - 1$

C8: n – 1

For this course, $\log_2 x = \log x = \lg x$

Better than Merge sort for smaller array sizes. It also uses less memory space than merge sort.

2.3 – Merge Sort

Demonstration with dance.
Gif Demonstration.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

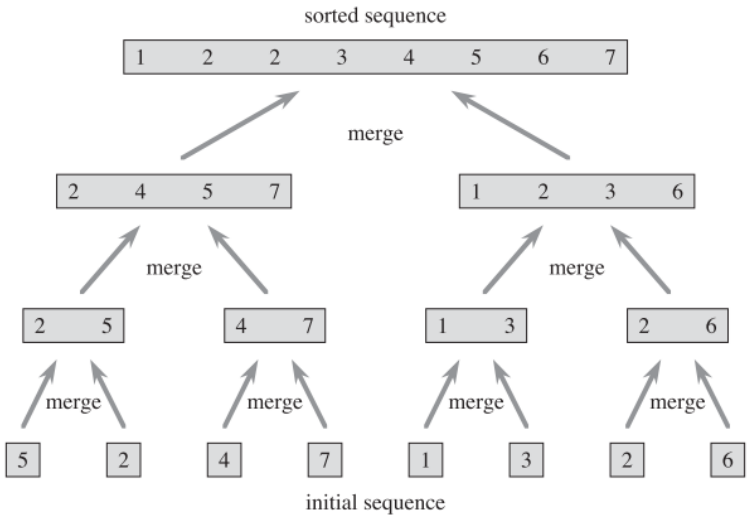


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

If the number is odd, split it...

Worst case = best = average = O(nlgn)
Slower than quicksort, until a certain *n*, (1billion elements?).

Auxiliary Space: space required to run an algorithm

Merge sort is quite memory intensive because it is not an **in-place sort**; merge sort requires $\Theta(n)$ space for all its copies of the array, while in-place sorts require $\Theta(1)$ space, since they don't make copies of the array.
Insertion sort is an in-place sort.

Recurrence Relationships

The general equation for a recurrence relationship is:

$$T(n) = \begin{cases} \Theta(1) & n \le c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), & \text{else} \end{cases}$$

- The top line defines when the run time of a function is linear. For example, in merge sort, the run time will always be the same when the size is 2. So *c* is 2.
- *a* is the number of divisions per recurrence
- *b* is the size of the next recursion
- $C(n)$ is the time it takes to combine each sub-problem
- $D(n)$ is the time it takes to divide each sub-problem

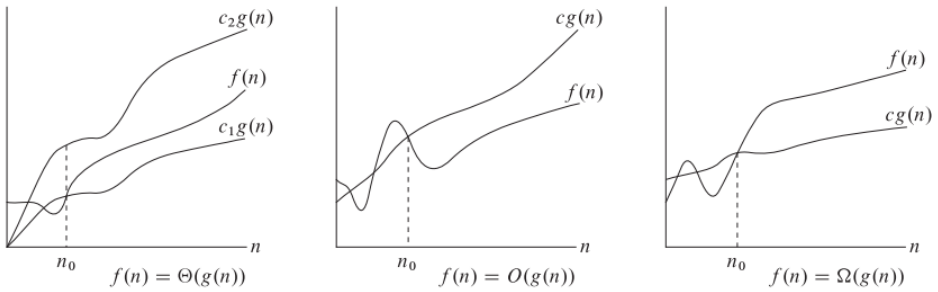
Chapter 3

Asymptotic notations

Asymptotic bounds: the behaviour of a function as functions get very large and approach infinity

Note: disregard any constants, since they become negligible as functions get very large

- Big Omicron: O ≤
- Big Theta: Θ =
- Big Omega: Ω ≥
- Small o: o <
- Small omega: ω >

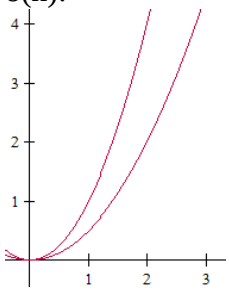


$$o(g(n)) = f(n)$$

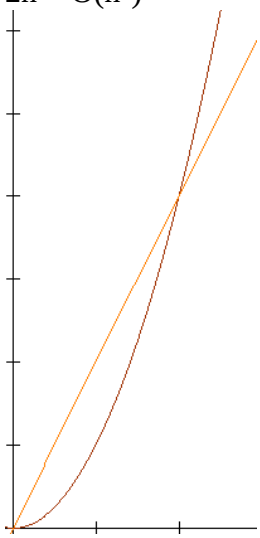
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

g(n) is going to be much bigger than f(n)

o(n):



$2n = O(n^2)$



for ω(g(n) = f(n),

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Asymptotic Upper Bounds

Page 47: given a function g(n)

$O(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.

$$g(n) \in O(g(n))$$

$$f(n) \in O(g(n)) \Leftrightarrow f(n) = O(g(n))$$

Generally, we focus on the O(n) because it models how the algorithm increases

If you want to find out information about an algorithm, use the Ω

Page 53: **Monotonicity**

Monotonicity

A function $f(n)$ is ***monotonically increasing*** if $m \leq n$ implies $f(m) \leq f(n)$. Similarly, it is ***monotonically decreasing*** if $m \leq n$ implies $f(m) \geq f(n)$. A function $f(n)$ is ***strictly increasing*** if $m < n$ implies $f(m) < f(n)$ and ***strictly decreasing*** if $m < n$ implies $f(m) > f(n)$.

Strictly: can only be going up or down

To figure out how long an algorithm takes, find the number of iterations.

If you are given $a + \frac{b}{n} + \frac{d}{n^2} \leq c$, multiply both sides by n^2 to get: $an^2 + bn + d \leq cn^2$ (i.e. get rid of fractions)

O(n) is sometimes only true between a given range. For example, $f(n) = 3x^2 - 5x + 4$, $O(f(n)) = x^2$, but only between …(?)

Chapter 4: Divide and Conquer [the world? no.]

Maximum sub-array problem

4.3 - Substitution Method

e.g. Determine upper bound on recurrence: $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$ (e.g. 4.19 page 83)

$T(n) = O(n\lg n)$

$T(n) \leq cn\lg n$

Assume holds for all positive $m < n$

$m = \lfloor \frac{n}{2} \rfloor$ = floor(n/2)

$T(\lfloor \frac{n}{2} \rfloor) \leq c(\lfloor \frac{n}{2} \rfloor)\lg(\lfloor \frac{n}{2} \rfloor)$, sub into the original, since it's a floor

$T(n) \leq 2(c(\lfloor \frac{n}{2} \rfloor)\lg(\lfloor \frac{n}{2} \rfloor) = n$

$\leq 2c(\frac{n}{2})\lg(\frac{n}{2}) + n$

$\leq cn\lg(\frac{n}{2}) + n$

$\leq \underbrace{cn(\lg(n))}_{\text{desired}} - \underbrace{c \cdot n + n}_{\text{residual}}, \quad \leq 0$

$c \geq 1$

residual ≤ 0

$\lfloor \frac{n}{2} \rfloor \leq \frac{n}{2}$

Root: node at top level of recursion tree

No parents

Batman
Bad Split: when one side of the tree is 0 and everything else goes on the other side of the tree
Good Split: when you have an even split

4.5 - Master Method

Page 94: **Master theorem**

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

a = # of sub-problems; n/b = size of the sub-problem; f(n) = base case, most of the time, it's O(1)
ϵ = used to make the exponent of t^n the same as that of f(n). Pg 94 paragraph under the cases

Compare $n^{\log_b a}$ to f(n) to decide which case you are going to use

- Case 1: $n^{\log_b a}$ > f(n)
- Case 2: $n^{\log_b a}$ = f(n)
- Case 3: $n^{\log_b a}$ < f(n)

e.g.

T(n) = 9T($\frac{n}{3}$) + n, a = 9, b = 3, f(n) = n

Choose ϵ = 1
$n^{\log_3 9} = \Theta(n^2)$, $f(n) = O(n^{2-\epsilon}), \epsilon = 1$
T(n) = $\Theta(n^2)$

e.g.

T(n) = 4T($\frac{n}{2}$) + $n^{\frac{7}{3}}$, a = 4, b = 2, f(n) = $n^{\frac{7}{3}}$

$n^{\log_b a} = n^{\log_2 4} = n^2$
Case 3: f(n) = $\Omega(n^{2+\epsilon})$, $\epsilon > 0$
f(n) $\geq c \cdot n^{2+\epsilon}$
$n^{\frac{7}{3}} \geq c \cdot n^{2+\epsilon}$
e.g. $\epsilon \leq \frac{1}{3}$, c = ½
Therefore, T(n) = $\Theta(n^{\frac{4}{3}})$

2)
$a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$

$4\left(\frac{n}{2}\right)^{\frac{7}{3}} \leq c \cdot n^{\frac{7}{3}}$

$2^{-\frac{1}{3}} n^{\frac{7}{3}} \leq c \cdot n^{\frac{7}{3}}$

$2^{-\frac{1}{3}} \leq c$

e.g. c = 1

e.g.

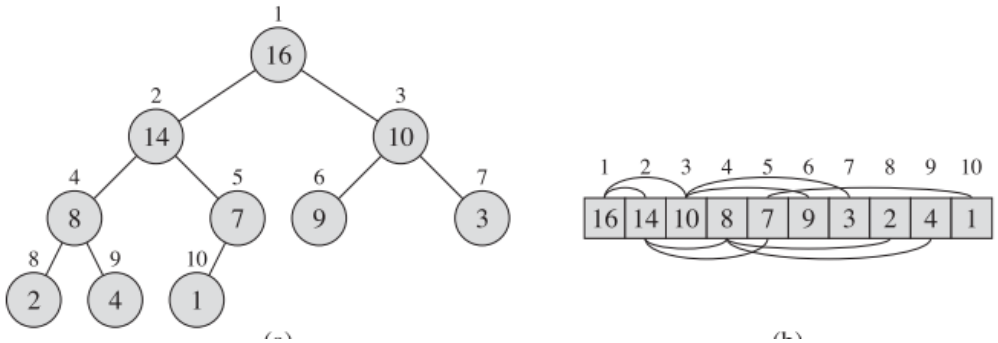
$T(n) = 2T(\frac{n}{2}) + n\lg n$, a = 2, b = 2, $f(n) = n\lg n$

$n^{\log_b a} = n^{\log_2 2} = n^1$

Looks like $n\lg n = \Omega()$
But for any $\epsilon > 0$, $= \omega(n\lg n)$
So $n\lg n$ asymptotically larger than n
BUT not *polynomially* larger (between case 2 & 3)
Thus, does not fit in the Master method.

Chapter 6: Heaps

Page 152:



Parent (i) = $\lfloor \frac{i}{2} \rfloor$
left(i) = 2i
right(i) = 2i + 1

Useful for priority queues

Root might not be larger than its children.
Max-Heapify puts the root in its place

Heap

- A.K.A. array tree

- Binary tree in the sense that each node has 0-2 children and 1 parent and an example implementation of a priority queue
- Implements a set, *S*, of elements associated with a key.
- Operations: pick max/min priority, delete, insert, change priorities
- height = lg*n*; min for a heap
- max for a heap is 2h
- root = I, left child = 2i, right child = 2i + 1
- key of a node = S[node]
- Cost: O(*n*lg*n*)
- n−1 branches; n−1 comparisons

2 types: minimum or maximum heap

- minimum heap has smallest number at the top
    - biggest value is a leaf that doesn't have to be at the bottom row
    - Minimum # of elements: 2^n
    - Maximum # of elements: $2^{n-1} - 1$
- maximum heap has the biggest number at the top

methods:

- insert(S,x) = inserts element, x into set, S
- max(S) = returns value of largest key, usually means highest priority in S
- extract-max(S) = max(S) + deletes it from S
- increasekey(S,x,k) = set key of S(x) to a a high

You can only delete the maximum value in a max-heap

Max-heap: key ≥ keys of children
max-heapify(A,i) = sorts heap into a max-heap
run time = O(lg*n*) for each iteration
so O(*n*lg*n*) for the whole tree

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Fully-filled in tree has 2^n nodes, where n = the level

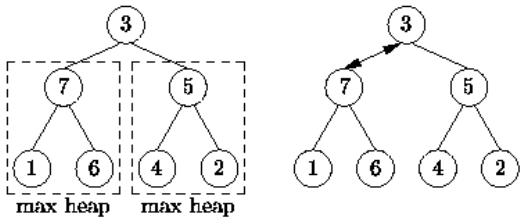
Array, A

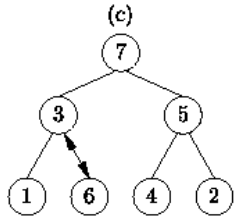
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

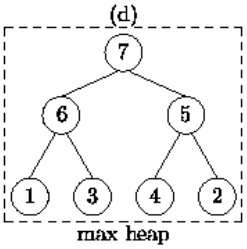
We defined heap size as the array length.

Example of heap-sort.

Example of Max-heapify.







Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)

Heap sort: running Max-Heapify on each level

Heap Property:

Priority Queue

Heaps can be used as something called a priority queue, where the maximum node is extracted each time and each node is updated with different priorities when it is inserted.

Chapter 7: Quicksort

Demonstration through a dance.
Quicksort gif.

Pointer: address of a particular variable/byte

Pivot point: point in the middle

Best case: splitting into 2 equal sections

To left of pivot is left partition; right, right partition.

You have to change the pivot each layer.

Run time: O(nlgn)
Worst: $O(n^2)$

Faster than merge sort below one billion elements.

$$\begin{array}{ccccccccccc} 1 & + & 2 & + & 3 & + & \dots & + & n \\ n & + & (n-1) & + & \dots & \dots & \dots & + & 1 \end{array} = \frac{n(n+1)}{2} = \Theta(n^2)$$

$$\underbrace{(n+1) \qquad\qquad (n+1)}_{n} \Big\} \frac{n(n+1)}{2}$$

Balanced?

Worst case is when it's in decreasing order because the default pivot is always on the right-most element.

$$T(n) = \underbrace{T(0)}_{\text{finding pivot}} + \Theta(n) + \underbrace{T(n-1)}_{\text{alg for every other element}} + \overbrace{\underbrace{\Theta(n-1) + \Theta(n-2) + \dots}_{\text{for each element}}}^{=\Theta(n^2)}$$

Chapter 10: Elementary Data Structures

Array: no book-keeping info

Stacks: LIFO (Last-In, First-Out)
Analogy: when you clean your dishes, say, plates, the first plates you'll pick up when it comes time to use them will be the top one, the most-recently washed one.
Analogy 2: a stack of cups at a fast food restaurant

Queue: FIFO (First-in, First Out)
Analogy: line-up
Head: First element of a queue
Tail: last element in a queue

predecessor: element in list with the next lowest value
successor: element in list with the next highest value

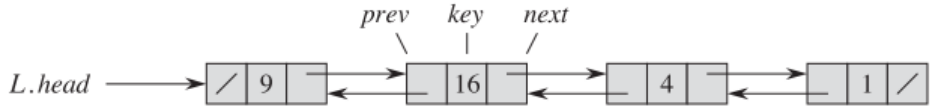
Stack under/overflow are for stacks & queue's
Stack underflow: empty stack, but command been sent
Stack overflow: call a location over the size of the stack

Linked List:
Data structure where objects arranged in linear order that is determined by pointer in each object

Sometimes, they contain a dummy first and last item, called the **sentinel**, which connects the head and tail to form a circular doubly-linked list. These help with ignoring boundary conditions at the head and tail of the list

Linked list vs. doubly-linked list: doubly-linked list has a pointer. What this does

Doubly-linked list (figure 10.3)



Problem 10-1:

	Unsorted, singly linked	Sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
Search(L,k)	O(n) gotta search thru whole array of elements	O(n)	O(n)	O(n)
Insert(L,x)	O(1)	O(1)	O(1)	O(1)
Delete(L,x)	O(1) Can remove element and point to the one after it	1	1	1
Successor(L,x)	1 Just follow the link	1	1	1
Predecessor(L,x)	n There is no link back in singly linked. Must search entire list.	n	1 There is a link back.	1
Minimum(L)	n Isn't sorted, have to search whole list.	1 Assumed to be sorted lowest→highest, first element is lowest	n	1
Maximum(L)	n Have to search whole list.	n Know where it is, but have to follow path there.	n Have to search whole list.	1 Know where it is, can jump right to it?

The tree is stored in an adjacency list, which gives us the order to iterate from; s is the first element in the adjacency list.

The chain just outlines which nodes the given node is connected to, i.e. the first line does not mean that 2 is connected to 5. The order is determined by the algorithm (which we shall find out)
1 []→[2|]→[5|/]
2 []→[1|]→[5|/]→[3|]→[4|]
3 []→[2|]→[4|/]
4 []→[2|]→[5|]→[3|/]
5 []→[1|]→[2|]→[4|/]

The graphs can be represented by a matrix, which also outlines the weight of each connection:

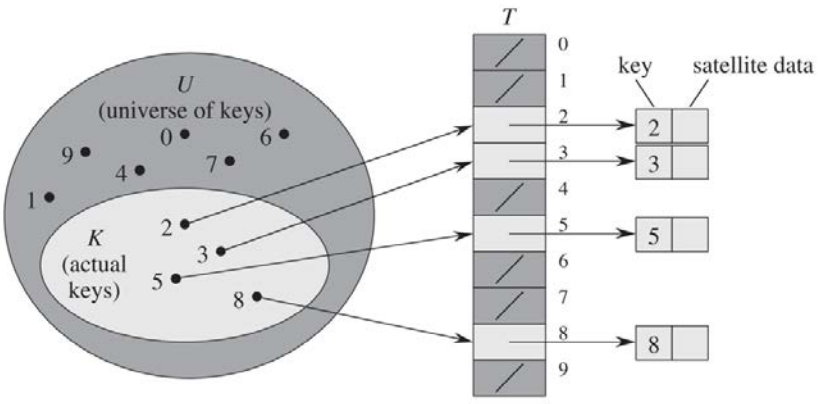
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0

Chapter 11: Hash Tables

Key: pointer to the location of an object

Direct Addressing: having a different key for each element in the universe, U of keys; impossible for large universes because it is so difficult to look through each key to find something. This is more useful when the set of non-empty slots, K, is not much smaller than U

Direct-address table: each position in the table corresponds to a key in the universe



Hashing: a.k.a. hash table, hash map; keys are values at given points, e.g. storing songs, key would be song name;

Chaining: type of hashing; when the end of the first array is "chained" to the beginning of a another array; pointers necessary to point to the next address; worst-case searching time for hashing with chaining is Θ(n); you need this because you don't know how many elements you are going to have at each entry

Types of hashing:

- Division Method: group by remainder, i.e. $h(k) = k \bmod m$
- Multiplication Method: $h(k) = \lfloor m(kA \bmod 1) \rfloor, 0 < A < 1$
- Universal hashing: random hash function

Open Addressing

Open Addressing: an attempt to eliminate collisions altogether by implementing a hash table with an array; no chaining; no pointers

No 2 values can be at the location. However, multiple values can share the same hash function value. How does this work? When inserting each value into the table: Insert (k,v)

1. The hash function computes which slot the data *should* go into.
2. Check if that slot is empty (a.k.a. **probing**)
3. If the slot is not empty
    a. Increase the **trial count**.
    b. Compute a new slot. Before I explain what that means, note that there are 2 values that go into the hash function: the key and a number called **trial count**.
    c. Go to step 2
4. Insert the key in the slot.

Probing: trying to find an empty slot for a key using the hash function. Why trying? Multiple values can share the same slot in these hash tables.

Trial Count: a number that identifies how many failed probe attempts were required to find a slot where the key fits. Think of it like different versions of the computed key. The trial count ranges from 0 to $m - 1$, where m is the size of the table

Types of probing:

- Linear
- Quadratic
- Double-hashing

Linear: where h'(k) is given, $h(k,i) = (h'(k) + i) \bmod m$

Quadratic: where 2 constants are also given, $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

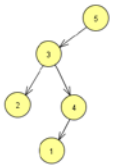
Double-hashing: a method of open addressing that uses 2 auxiliary hash functions to further randomize values, $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$

$\delta(u,v) \leq \delta(s,u) - \pi$

Chapter 12: Binary Search Trees (BSTs)

Things to the bottom left of the node are less than it;
things to the bottom right of the node are greater than it.

Notice how there is a leaf to the right of the left-most leaf that is less than it:



Chapter 15: Dynamic Programming (DP)

Dynamic Programming

- "careful brute force"
- **Memoize** and reuse solutions to sub-problems that help solve original problems

It's a way to get away from greedy algorithms. It can be done with bottom-up filling in of the table, but is usually top-down or **memoization**; both have linear time

Focus on the Fibonacci example, instead of the rod-cutting example, which is super confusing

Memoize: remember each step of the recursion

Naive Recursive

$\underline{\text{fib}}(n)$:
    if $n \leq 2$: return $f = 1$
    else: return $f = \text{fib}(n-1) + \text{fib}(n-2)$
$\implies T(n) = T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n$
    $\geq 2T(n-2) + O(1) \geq 2^{n/2}$
    EXPONENTIAL — BAD!

Methods:
1) Memoized
    memo = {}
    fib(n):
        if n in memo: return memo[n] ← if key is already in dictionary, return value (do not compute again)
    if n ≤ 2: f = 1
    else f = fib(n − 1) + fib (n − 2)
    memo[n] = f
    return f

- fib(k) only recurses first time it's called
- memoized calls take Θ(1) (basically free)//I think they would still take Θ(n) because you need to search a memoize table for them, but that's better than 2^n
- # of non-memoized calls is n: fib(1),...,fib(n)
    - non-recursive work per call = Θ(1)

Run-time = (# of subproblems) · (time/subproblem)

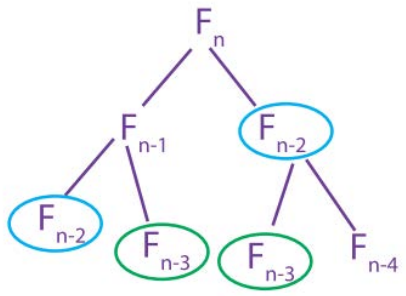


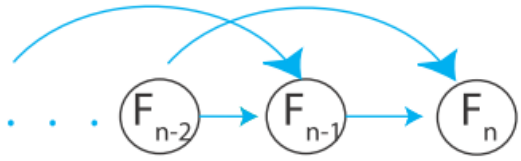
Figure 1: Naive Fibonacci Algorithm.

For the memoized algorithm, you'd stop going down on the blue circle on the right, so you wouldn't even see the right green circle

Same! i.e. already in dictionary. Therefore left continues; right stops

2) Bottom-Up DP

```
fib = {}                                          } Θ(n)
for k in [1, 2, ..., n]:                          }
    if k ≤ 2: f = 1                               }
    else: f = fib[k − 1] + fib[k − 2]   } Θ(1)     }
    fib[k] = f                                    }
return fib[n]                                     }
```



Same as method 1, different wayto think about it, often more efficient (saves space, only need to remember last 2 numbers)
Topological sort of subproblem dependency DAG

In summary: do basic stuff first that helps you compute the complicated stuff later on

Shortest Paths

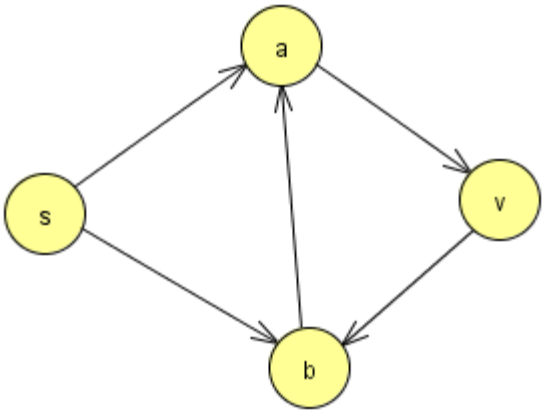
δ(s,n) ∀ v
− Guess: try all guesses, pick best one
e.g. shortest path from s to v

want shortest $s \to v$ path

- last edge must come from u, try all
- δ(s,n) = min(δ(s,n) + w(u,v)) ← will choose best/shortest path
- takes −∞ time with negative weight cyclic graphs
- ignores a positive weight cycle
- DAG: O(V + E)

time = indegree(v) + 1 ← in case indegree(V) = 0
total time ($\sum_{v \in V}$ indegree(v)) + 1 = O(E)

Subproblem dependences must be acyclic
Why?
e.g. δ(s,s) = 0



$\delta(s,v) - \delta(s,a) - \delta(s,b)$ $\begin{matrix} \delta(s,s) \\ \delta(s,v) \end{matrix}$ ← what we are trying to find

5 Basic Steps:

1. Define subproblems − # of subproblems
2. Guess part of the solution
3. Relate subproblem solutions with recurrence
4. Build algorithm − check subproblem is acyclic (has topological order)
5. Solve original problem

Examples	Fibonacci	Shortest Paths
1) Subproblem:	Fk for k = 1,...,n	δk(s,v) for vϵV,

# of subproblems:	n	$0 \le k \le \|v\|$, V^2
2) Guess: # of choices	Nothing 1	edge info v (if any) indegree(v) + 1 ← when no edges occur
3) Recurrence: Time/sub-problem:	$F_k = F_{k-1} + F_{k-2}$ $\Theta(1)$	$\delta_k(s,v) = \min\{\delta_{k-1}(s,n) \mid (u,v) \in E\}$ $\Theta(\text{indegree}(v) + 1)$
4) Topological Order: time:	k = 1,...n $\Theta(n)$	for k = 0,1,...$\|v\| - 1$ for vϵV $\Theta(VE)$
5) Solve: extra time:	F_n $\Theta(1)$	$\delta_{\|v\|-1}(s,v)$ $\Theta(v)$

e.g. Text Justification

split text into "good" lines (align nicely)
text = list of words

$$\text{badness}(i,j) = \begin{cases} (\text{pagewidth} - \text{total width})^3 & \text{if fits in line} \\ \infty & \text{if doesn't fit in line} \end{cases}$$

→ use words [i, j] as line

1) Subproblems: # subproblems:	suffixes words [i,j] n
2) Guess: Choices:	where to start 2nd line $\le n - i = O(n)$
3) Recurrence:	DP[i] = min(DP(j) + badness(i,j)) time/subproblem = O(n) for j in range (i + 1, n + 1): • i is first word of first line • j is first word of second line • DP(n) = 0
4) Topological order: Total time:	i = n, n − 1,...0 $\Theta(n^2)$
5) Original problem gets solved	DP(0)

Parent Pointers: remember which choice was best

e.g.

Perfect-Information, Black-jack

- deck = $c_0, c_1,...,c_{n-1}$
- 1 player vs dealer
- 1 bet/hand

1. Sub-problems: cards remaining

2. Guess: how many times should I hit per hand?

1) Subproblems: # subproblems:	Suffix c_i n
2) Guess: # choices:	How many times hit? $\leq$ n
3) Recurrence:	BJ(i) = max(outcome ϵ {−1,0,1}←$ earned / lost +BJ(j)→j = [i +4+#hits + #dealer hits] for #hits in range (0,n) if valid play)

Sub-problems for strings / sequences

suffixes $x[i:]$ } $\Theta(|x|)$

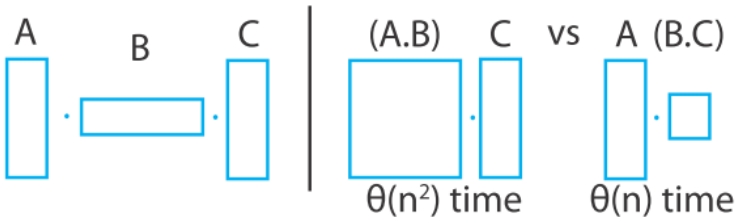
prefixes $x[:i]$

substrings $x[i:j]$ } $\Theta(x^2)$

Note: never use both prefix + suffix in same sub-problem (use substring instead)

Parenthesization

Optimal evaluation of associative expression $A[0] \cdot A[1] \cdots A[n-1]$ — e.g., multiplying rectangular matrices



Θ(n), therefore linear, better parenthesization
For associative expressions it doesn't matter where parameters are, but some will be "cheaper" than others. For example, matrix multiplication.

1) Subproblem: optimal evaluation of $A_k...A_{j-1}$
subproblems: n^2
2) Guess: outermost / last multiplication
e.g. $(A_0...A_{k-1})\cdot(A_k...A_{n-1})$
prefix suffix
$(A_0...A_{k-1})\cdot(A_k...A_{k-1})$
substring
$(A_j...A_{k-1})\cdot(A_k...A_{j-1})$
#choices: $O(j - i + 1) = O(n)$
3) Recurrence: DP(i,j) = min(DP(i,k) + DP(k,j) + cost($A_{i:k}$)($A_{k:j}$) for k in range(i+1,j))

time/subproblem: O(n)
4)Total time: $\Theta(n^3)$
Topological: Increasing substring size

Chapter 16: Greedy algorithms

Think of booze. E.g. At a party with free booze. If you want to get drunk fast drink whatever's in front of you.

Elements of a greedy algorithm:
1. Greedy choices made so far are part of a universal, optimal solution, S (*greedy choice property*).
2. The optimal solution to the sub-problem is part of a universal, optimal solution, S (*optimal substructure*).
3. Can be a recursive *or* iterative solution

Imagine that you're robbing a gold vault and trying to fill your backpack with soft gold bars; edge is item, node is state
Fractional Knapsack: you may pick up parts of items, thus you look for the item with the highest money:weight ratio
0-1 Knapsack: only pick the whole items that fit; take it or leave it − no middle way, Buddhist crap; think of it like binary

Greedy: you always pick the most valuable item, the shortest edge, so you're greedy, like the thief in the gold vault

Theorem 16.1
Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Stupid greedy algorithm: according to your requirements, you go for the higher priority; very dumbed-down example of what a greedy algorithm is; for example, looking for the best money:weight ratio, which isn't always optimal, but can be

If you have negative weights and you can create a cycle that has a net negative weight (*negative cycle*), you can have a net weight as negative as you want.

Chapter 22: Elementary Graph Algorithms

Adjacency Lists & Matrices

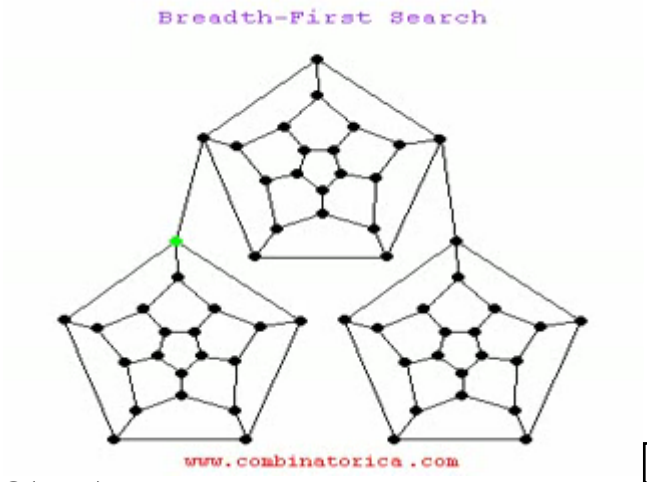
$$A = a_{ij} = \begin{cases} 1, & (i,j) \in E \\ 0, & \text{else} \end{cases}$$

Degree: the degree of a vertex in an undirected graph is the number of edges incident on it
Isolated: vertex with degree 0

Outdegree: number of edges leaving a vertex of a directed graph
Indegree: number of edges entering a vertex of a directed graph

BFS

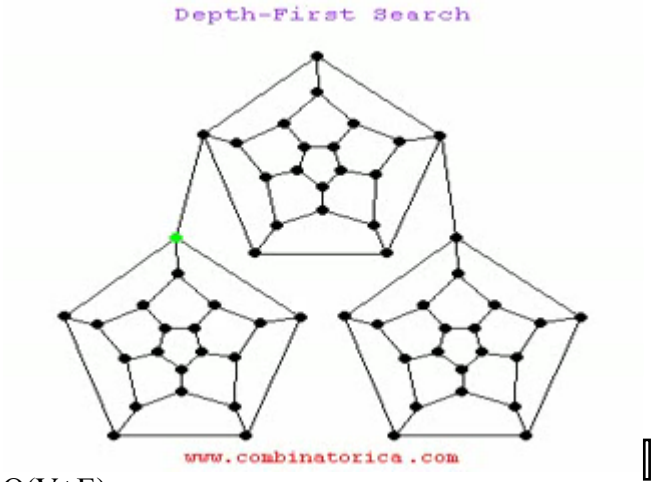
BFS (Breadth first search) − undirected graphs



O(V+E)

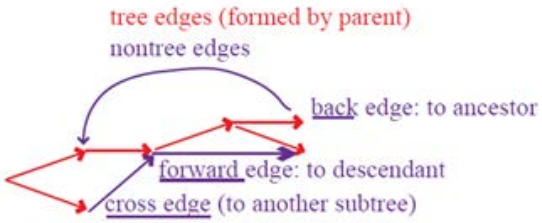
DFS

DFS (Depth first search) − directed graphs



Θ(V+E)

Types of graphs:
-Directed - Can only travel in the direction of the pointer
-Undirected - Can travel through any line any direction



1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

Acyclic: no cycles

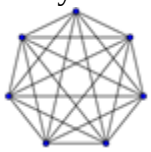
Strongly-connected component (SCC): a part of a directed graph where there exists a path to get from every vertex to every other vertex

Multi-graph: graph with multiple edges between a pair of vertices; can have self-loops. There exists both directed and undirected multi-graphs.

Chapter 23: Minimum Spanning Trees

Spanning tree: acyclic; connects all vertices; not necessarily a complete graph

Complete graph: everything connected by unique path; all the nodes are directly connected to every other node.



bottleneck: heaviest edge in a tree

Minimum Bottleneck Spanning Tree (MBST): the tree in a graph with the lowest bottleneck; no cycles; linear time to find an MBST; came up in tutorial

Safe edge: when connecting multiple trees, adding it will result in a new forest that is STILL a part/subset of an MST; Invariant, A, is part of a bigger MST

Graph, G(V,E) ← V = set of vertices; E = set of edges
edge(u,v), where u = , and v = vertex
Digraph = directed graph

Prim's and Kruskal's Algorithms are greedy algorithms that find a minimum spanning tree for a connected, weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Both run in $O(E \lg V)$ if you use binary heaps.

Kruskal's algorithm: greedy algorithm that finds minimum spanning tree that is both iterative and recursive methods; 2 nodes become 1; choose lightest edge in the whole graph, then next lightest (doesn't have to be connected as you are working through it as long as it joins up in the end), until all vertices are connected; looks at the smallest weight on the graph and draws a edge, if the nodes have already been "explored" and the edge is useless it skips it, it does this till every node has been "explored"

Kruskal

T = Kruskal(G) //Greedy Algorithms
If G = v, then return v,
Find min weight edge, e. (1)
Contract e à G' = G \ {e} /*e = weight of the edge. By definition, removing the weight of an edge means removing the edge itself*/
T' = Kruskal(G') (2)
T(n+m) = (1) + T(n−1, m−1)
Return (T' + e)

Prim's minimum spanning tree algorithm: a greedy algorithm; choose any starting vertex and look for edge with lowest weight and add it to the tree. Next, choose edge with lowest weight connected to the tree. This tree spans all vertices, so continually repeat this until you've covered them all; similar to Dijikstra's

[Trees] Cut: the splitting of the nodes in a graph into 2 parts to get 2 minimum-spanning trees. If there are any cycles, you can use it to break up the cycles. It is also useful if you know that the item you are looking for is in a specific part of a big tree, so you don't have to spend as much time looking through the whole tree. After cutting a minimum spanning tree, the lightest edge (weight) is a safe edge; notation for a cut (S, V−S), where S = the connected nodes. A set of edges, A, is a forest, s.t. , T, that satisfies the invariant

Summary:
Kruskal's adds one edge at a time. The next line is always the shortest (minimum weight) ONLY if it does NOT create a cycle.

Prim's adds one vertex at a time. The next vertex to be added is always the one nearest to a vertex already on the graph.

Chapter 24: Single-Source Shortest Path

Shortest path from s to every other node, but edges are weighted with no loss of generality.
Shortest path weight from u to v: δ(u,v)

Negative weight edges are a problem. None can deal with negative-weighted cycles.

Can a graph have a cycle at all for a shortest path? If no weight, all positive, simple path.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Relaxation

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Line 2: You realize there's a shorter path than your estimate

There are 6 properties of shortest paths and relaxation: (page 650)

Triangle inequality (Lemma 24.10)
For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 24.11)
We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 24.12)
If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 24.14)
If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 24.15)
If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Predecessor-subgraph property (Lemma 24.17)
Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Predecessor: once all estimates are correct, you have a subgraph that is all shortest paths from s.

24.1 Bellman-Ford

More a checker for shortest path than a finder. Also checker for negative cycles.

O(V·E)
E = O(V2)
Therefore, O(V3)

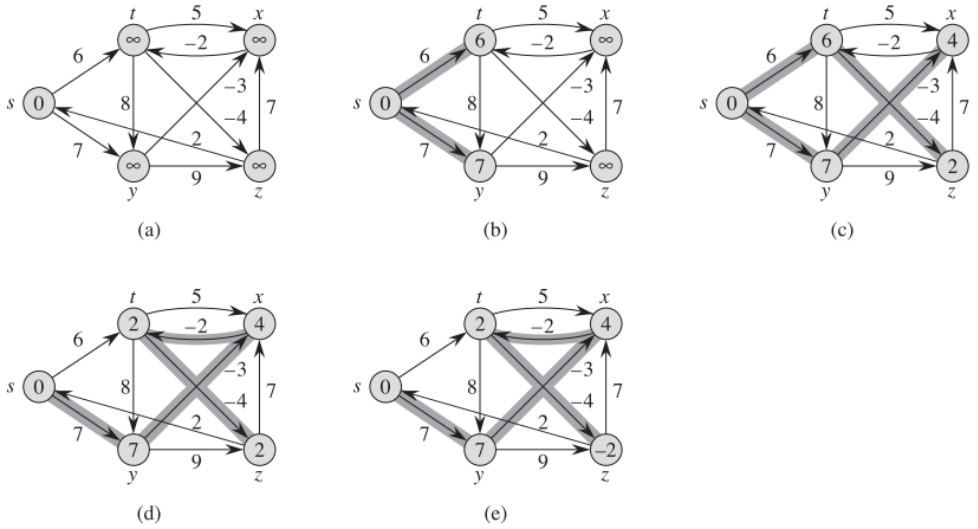
Page 651: Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Line 2: iterate V − 1 times (where huge run time comes from)
Line 5-7: Checks if negative weights. If negative weights, then return false.

Once you relax ALL your predecessors, later relaxations WILL result in a shortest path. Relax everything, everything times. Each edge is being visited V−1 times.

e.g. Page 652



Explanation: It's brute force and isn't greedy, so even if you have the most relaxed possible value, you still have to relax it, until the end even though nothing will happen to the edges that are already good. It doesn't necessarily start where you want it to start. First you do correctness distance 1, then correctness distance 2, etc, like a BFS.
a) Initial
b) First iteration of relaxing edges; correctness level 1

$\delta(s,t) = 6$, but $t.d = \infty$, so make $t.d \leq 6$

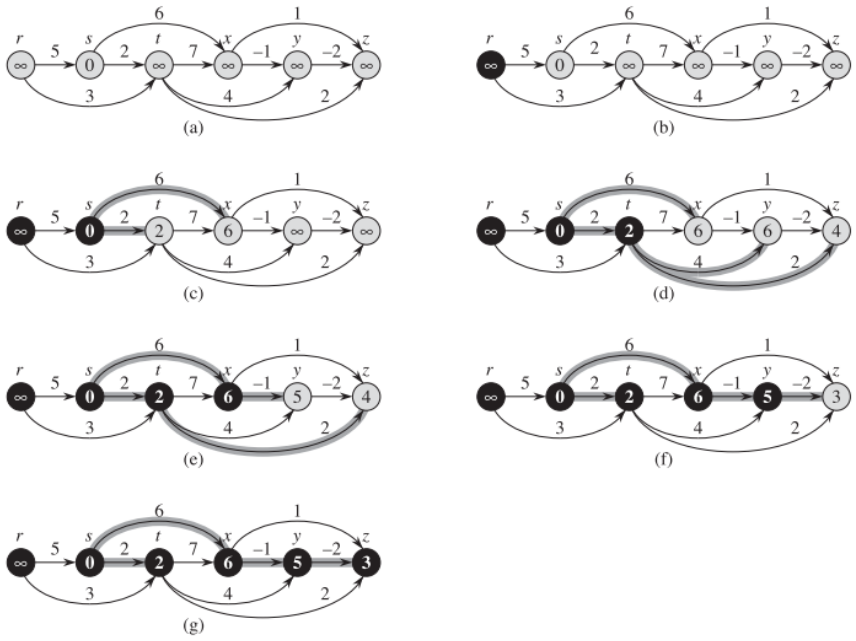
c) Correctness level 2
d) Iterating through all the edges
e) Finished!

24.2: Single-Shortest path in DAG's *woof*

You know what order because you have sorted, so it is a lot faster, since you don't ever move backwards.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       for each vertex v ∈ G.Adj[u]
5            RELAX(u, v, w)
```

e.g. page 656



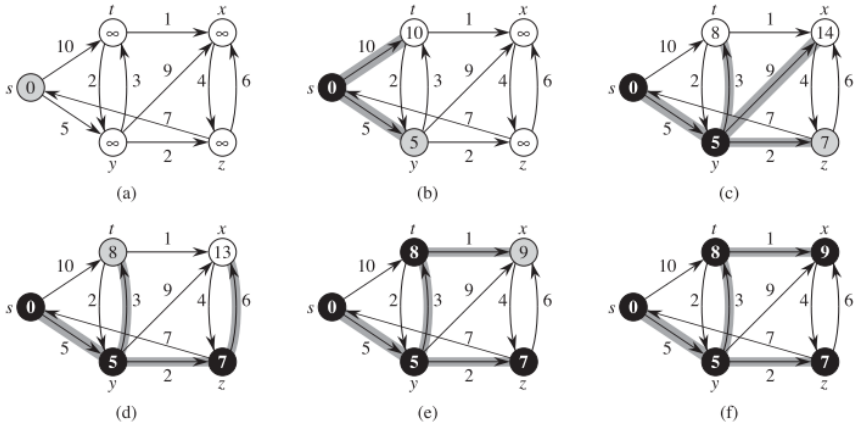
a) given graph
b) Select single source
c) Relax
d)
e)
f)
g)

24.3: Dijkstra's

- NO negatively-weighted edges
- Uses min-heaps, probably a binary heap because it uses minimum priority queues
- brute force
- More inituition, than proof
- The top node of your priority queue is always correct at the time of extraction (i.e. taking the top node of the heap)

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

e.g. Page 659



Black: node has been extracted
Grey: current node

a) Start!
b) Extract minimum node
c)
d)
e)
f)

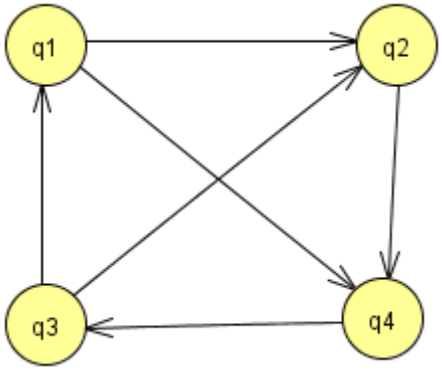
Chapter 25: All-Pairs Shortest Path

25.2: Floyd-Warshall

Floyd-Warshall shortest path algorithm: get from every vertex to every other vertex. There are multiple implementations. It is the bottom-up filling in of the table. There can be no negative cycles (although negative edges are OK). The predecessor matrix is Π. The weight matrix is D. $O(V^3)$

Page 694:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1. \end{cases} \qquad (25.5)$$



$$D^{(0)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 4 & \infty \\ \infty & 0 & b & 5 \\ \infty & \infty & 0 & 7 \\ 9 & \infty & \infty & 0 \end{pmatrix} \end{array}$$

$\min\left(d_{4,3}^{k-1}, d_{4,1}^{k-1}, d_{1,3}^{k-1}\right)$

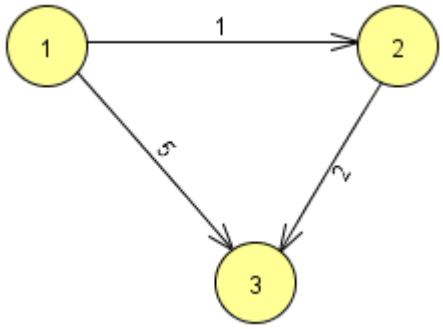
Since no negative edges, all edges are carried over to the new one
When k = 3, you may use the vertices numbered 1-3. So in the $D^{(1)}$, we are only allowed going through vertex number 1.

Transitive closure between a and b means there is path between the two vertices
Transitive relationship: whenever A > B and B > C, then also A > C

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty. \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

If the new path is shorter, then change it. If not, don't.



All 0's and ∞'s are NIL. Let's represent NIL by just an "N".

$$\Pi^{(0)} = \begin{pmatrix} N & 1 & 1 & N \\ N & N & 2 & 2 \\ N & N & N & 3 \\ 4 & N & N & N \end{pmatrix}$$

$$\min\left(d_{ij}^{(k-1)}, +\right)$$

$$\min\left(\infty, 9+4\right)$$

$$\Pi^{(1)} = \begin{pmatrix} N & 1 & 1 & N \\ N & N & 2 & 2 \\ N & N & N & 3 \\ 4 & 1 & 1 & N \end{pmatrix}$$

The weight diagonal is always 0, i.e. $D^{(k)} = \begin{pmatrix} 0 & & & \\ & 0 & & \\ & & 0 & \\ & & & 0 \end{pmatrix}$

The predecessor diagonal is always NIL, i.e. $\Pi^{(k)} = \begin{pmatrix} N & & & \\ & N & & \\ & & N & \\ & & & N \end{pmatrix}$

$$\min\left(4, 0+4\right), 4 \le 4$$

e.g. 2 Page 696

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

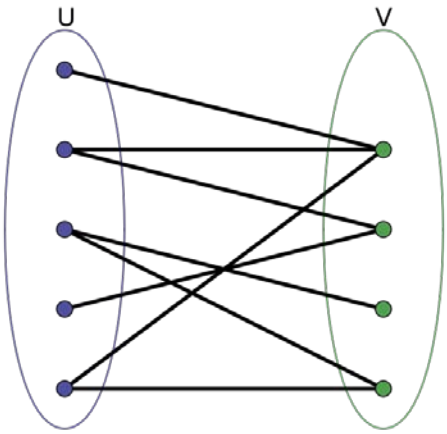
$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

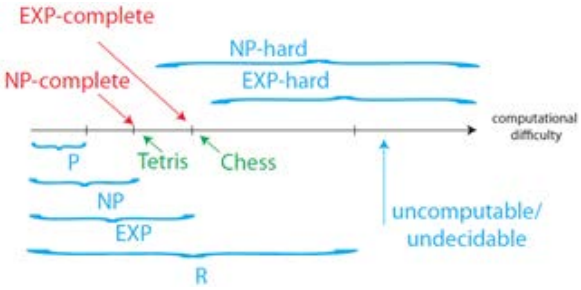
$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

e.g. 3)

Bipartite graph: 1-1 function



Chapter 34: NP-Completeness

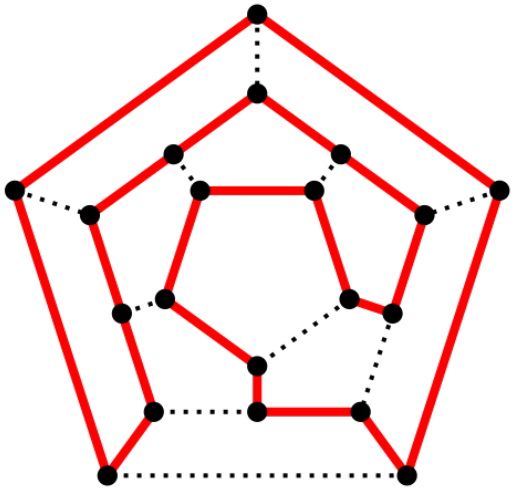


Languages in NP-hard are not verifiable in polynomial time, but are reducible.

Complexity Class P: problems solvable in polynomial time (run-time = $O(n^k)$)

P = $\left\{L \subseteq \{0,1\}^* : \text{ there exists an alg, A, that decides L in poly time}\right\}$

Complexity Class NP: {x $\subseteq$ {0,1}*: there exists a certificate, y, |y| = 0 ($|x|^c$), s.t. A(x,y) = 1}
e.g. Hamiltonian cycle: cycle covering all vertices



Verification Algorithm: L = {xϵ{0,1}*, there exists yϵ{0,1}*, s.t. A(x,y) = 1

A language (think in terms of SE 2FA3), L $\subseteq$ {0,1}* NPC:f:
1) L ϵ NP
2) L'$\leq_p$ L for every L'ϵNP

Polynomial reducibility:
ax + b = 0
ax^2 + ax + b = 0

$P \subseteq NP$
P = NP? P ≠ NP? Is NP is solvable in polynomial time?
NPC: NP-complete
examples of NP problems: circuit satisfiability (page 1070)

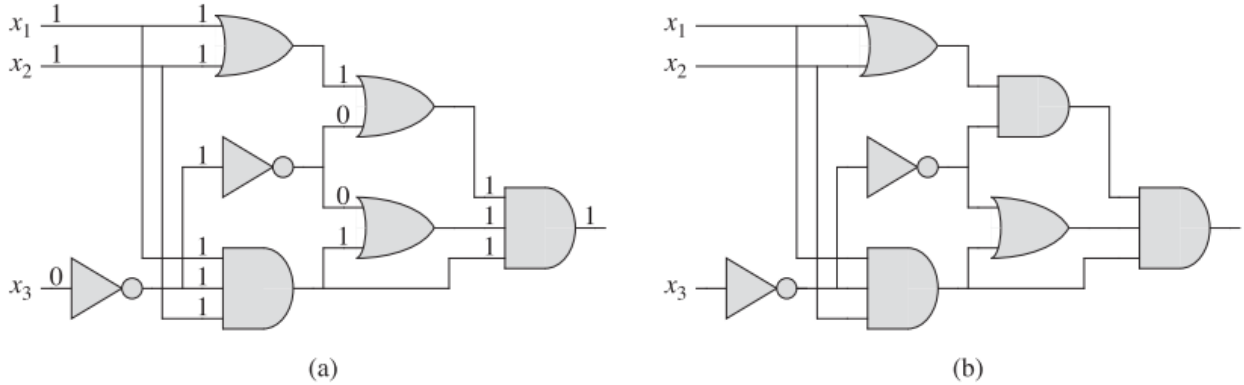


Figure 34.8 Two instances of the circuit-satisfiability problem. **(a)** The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

Proving NPC:
Find an alg, A, that can verify L in polynomial time.

NPC $\leq_p$ P
Proof [problem-specific]: (Page 1078)
Prove L ϵ NP
Select a known NPC-complete language, L'.
Find a poly-time reduction function, f, that reduces L' to L. algorithm, A, that verifies L in polynomial time.

length (x1,...xn) => O(n)

e.g. Formula satisfiability (page 1079)
SAT ≤p P

1) Circuit-sat is NPC
circuit {x ϵ {0,1}*, L satisfies the circuit
SAT{ϕ}: ϕ is satisfiable}

Page 1081

NP-Hard
Page 1069:
L ϵ NP
L' ≤p P for every L' ϵ NP

Co-NP: set of languages whose complement is in NP; it says yes whenever the NP checker says no.
Circle = language, red = co-NP, yellow = NP, orange = both
P = co-NP, P is in orange

MIPS - Microprocessor without Interlocked Pipeline Stages

Clique problem: finding particular complete sub-graphs ("cliques") in a graph, i.e., sets of elements where each pair of elements is connected.

Appendix B:

In a given digraph, an edge (u,v) is incident to or enters v and is incident from u.

Midterm:
Part I:
1)
3)

Part II:
2b)

15 17
14 12,13 11
9 7,4 18

Switch 12 and 18
Switch 18 and 15

2c) Make a new version of BUILD-MAX-HEAP
Recursion tree:
Height = logn
Width = n

Divide: RECURSIVE-BUILD(A) split into 2 sub-problems (first/last half of A)
Conquer: Recursion of RECURSIVE-BUILD until size 1
Combine: MAX-HEAPIFY between the sub-problems, height
height ,sub-problems → Total runtime: Θ()

3)

Part V:
Input: Adjacency-list
G = (V,E)
Output: starting soon

O(V + E)

Size of input: # one-way doors + 2 × # 2-way doors + # rooms
Use strongly-connected components
Topologically sort it
DFS on the very first node