1) Write the recursive relation for the running time of the algorithm given to solve 'Tower of Hanoi' problem and provide a solution for it.

$T(n) = 2T(n-1) + 1$

1) Base Case; $T(0) = 0 \quad T(1) = 1$

2) Inductive Case; Assume that $T(n) = 2^n - 1$,

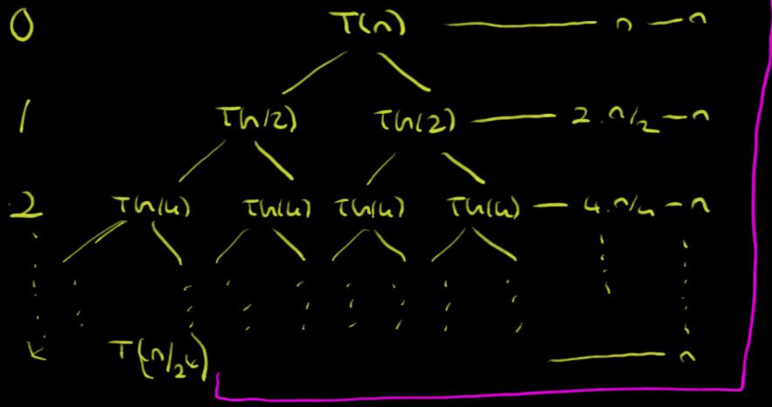
$T(n+1) = 2^{n+1} - 1$ ①

$T(n+1) = 2T(n) + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 1$ ②

① = ②

2) The running time for the merge sort algorithm is given as "T(n) = T(n/2) +T(n/2) +O(n)". Use a recursion tree, to show that T(n) = O(n.logn). Show all your work.

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



$n/2^k = 1, \; n = 2^k, \; k = \log n$

height $= k = \log n$

$T(n) = O(n.\log n)$

3) Give an example to a divide-and-conquer algorithm (recursive) to multiply two n-digit numbers with O(n^2) running time. Do you know any other algorithm that runs faster, meaning with o(n^2) running time? If yes, describe that algorithm and its running time as well.

Multiply 2 n-digit numbers less than $O(n^2)$ time?

Karatsuba Algorithms

$x.y = (10^{n/2} x_L + x_R).(10^{n/2} y_L + y_R)$

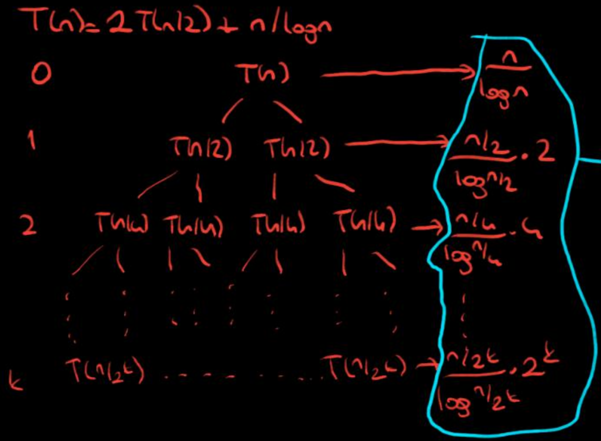
$= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \rightarrow$ 4 multiplication

$x_L y_R + x_R y_L = (x_L + x_R)(y_R + y_L) - x_L y_L - x_R y_R$ } 3 multiplication

$T(n) = 3T(n/2) + O(n)$

4) Consider the recurrence relation T(n) = 2T(n/2) + n/logn. Use a recursion tree, to show that T(n) = Θ(nloglogn).

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



$n/2^k = 1, \; \underline{k = \log n}$

$$\sum_{i=0}^{k} \frac{n}{\log n/2^i}$$

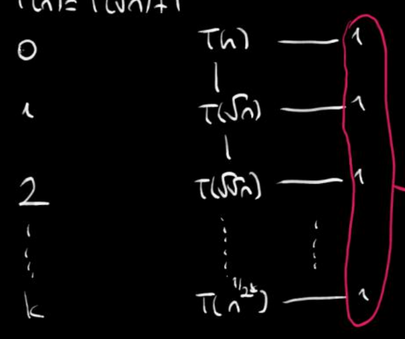
$$= n \boxed{\sum_{i=0}^{\log n} \frac{1}{\log n - i}}$$ Harmonic number $= \Theta(\log n)$

$= n \log n . k$ (height)

$\underline{= n \log\log n}$

5) a) Consider the recurrence relation T(n) = T(√n) + 1. Use a recursion tree to show that T(n) = Θ(loglogn).

$T(n) = T(\sqrt{n}) + 1$



$n^{1/2^k} = 2, \; 1/2^k = \log_n 2$

$, 2^k = \log_2 n, \; k = \log\log n$

→ $k.1 = k$

$T(n) = T(1) + k$

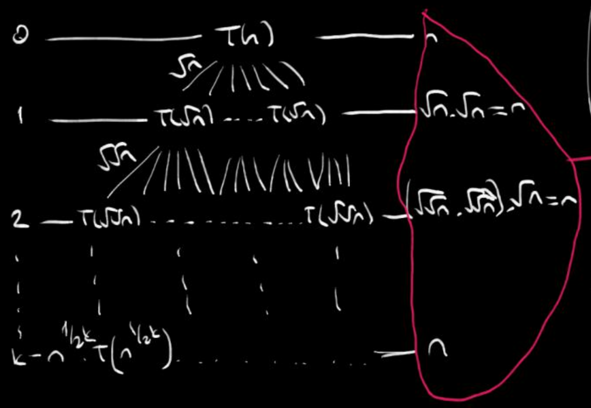
$\downarrow$

$\log\log n$

$\underline{T(n) = \Theta(\log\log n)}$

b) Similarly, if T(n) = √nT(√n) + n, show that T(n) = Θ(nloglogn).

$T(n) = \sqrt{n}\, T(\sqrt{n}) + n$



$n^{1/2^k} = 2, \; 1/2^k = \log_n 2$

$2^k = \log n, \; k = \log\log n$

→ $k.n$

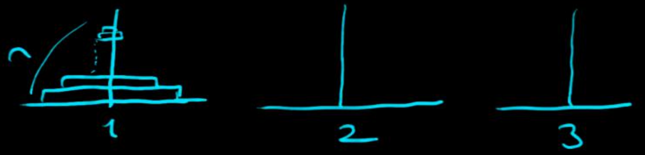
$T(n) = T(1) + k.n$

$= 2 + n \log\log n$

$\underline{T(n) = \Theta(n \log\log n)}$

7) Argue briefly why the recursive relation for the running time of the Tower of Hanoi problem is T(n) = 2T(n-1)+1 for n at least 1, in an recursive algorithm provided in class. In other words, explain why we cannot do better with a shorter running time or why not more running time is needed in the given recursive algorithm.

$T(n) = 2T(n-1) + 1$ (n diski 3. kolona taşımak istiyoruz.)



1. adım: (n-1) diski 2. kolona taşı. T(n-1)

2. adım: n. diski 3. kolona taşı. T(1)

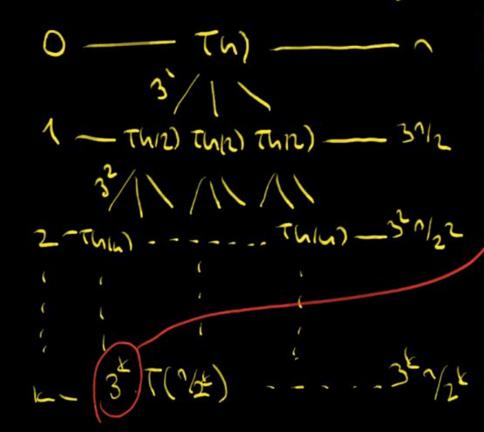
3. adım: (n-1) diski 2'den 3. kolona taşı. T(n-1)

} $T(n) = 2T(n-1) + 1$

Neden en iyi çözüm?

6) a) Karatsuba's multiplication algorithm has running time T(n), where T(n) = 3T(n/2)+n. Use a recursion tree to show that T(n) = Θ(n^log3).

Karatsuba $\Rightarrow T(n) = 3T(n/2) + n$



0 — T(n) — n
3
1 — T(n/2) T(n/2) T(n/2) — 3n/2
3²
2 – T(n/4) .... T(n/4) — 3²n/2²
k — 3^k T(n/2^k) .... 3^k n/2^k

3^k nodes,

each line $n/2^k$ value;

$$T(n) = \sum_{i=0}^{k} (3/2)^i \cdot n$$

Geometric Series, final term

$k = 3/2 \cdot n$ | $n/2^k = 1 \Rightarrow k = \log n = \text{height}$

So; $T(n) = 3^{\log_2 n} = n^{\log_2 3}$

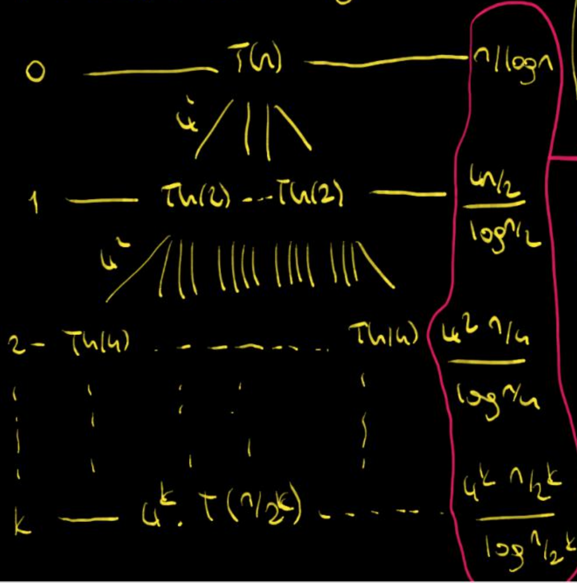
b) Let T(n) = 2T(n/2) +n/logn. Show that T(n) = Θ(nloglogn). --> Go to Question 4

8) a) Write a short recursive program to calculate Fibonacci number n.

```
int fibonacci(int n){
    if (n==1)
        return 1;
    else
        return fibonacci(n-1)+fibonacci(n-2)
}
```

c) Let T(n) = 4T(n/2) +n/logn. Show that T(n) = Θ(n^2).

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



0 — T(n) — n/logn
4
1 — T(n/2) ... T(n/2) — 4n/2 / log n/2
4²
2 – T(n/4) .... T(n/4) 4² n/4 / log n/4
k — 4^k . T(n/2^k) ...... 4^k n/2^k / log n/2^k

$n/2^k = 1$, $k = \log n$, 4^k nodes

$$\sum_{i=0}^{k} \frac{2^i \cdot n}{\log n/2^i}$$

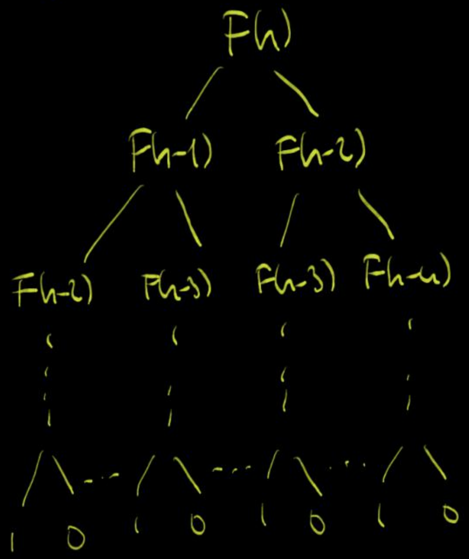
$$= n \cdot \sum_{i=0}^{k} \frac{2^i}{\log n - i} \quad (j = \log n - i)$$

$$= n \cdot \sum_{j=i}^{\log n} 2^{\log n - j} \cdot \frac{1}{j} = n \sum_{j=i}^{\log n} \frac{n}{2^j \cdot j}$$

$$n^2 \sum_{j=i}^{\log n} \frac{1}{2^j \cdot j} \sim 1 = \Theta(n^2)$$

b) In the recursion tree for this algorithm, give the number of leaves in this tree and prove that it is correct by induction.

$F(n) = F(n-1) + F(n-2)$



Son satırda → F(n) adet 1, → F(n-1) adet 0 } mevcut

Toplam = 2F(n+1) −1 node

sadece 1 adet root var.

Proof;

1) F(2), F(1), F(0) } 3 Node $2F(3)-1=3$

2) $F(n)=2F(n+1)-1$, $F(n+1)=2F(n+2)-1$

$F(n+1) =$ → F(n) adet 0, → F(n+1) adet 1 $\geq 2F(n+2)-1$ adet node

9) In the conflict-free class-scheduling problem, a) Prove that at least one maximal conflict free schedule includes the class that finishes first.

Let f be the class that finishes first. Suppose we have a maximal conflict-free schedule X that does not include f. Let g be the first class in X to finish. Since f finishes before g does, f cannot conflict with any class in the set S \ {g}. Thus, the schedule X' = X ∪ {f} \ {g} is also conflict-free. Since X' has the same size as X, it is also maximal.

To finish the proof, we call on our old friend, induction.

b) Prove that the greedy algorithm that picks the non-conflicting class that finishes first from the available classes outputs an optimal schedule (with maximal number of classes).

Let f be the class that finishes first, and let L be the subset of classes the start after f finishes. The previous lemma implies that some optimal schedule contains f, so the best schedule that contains f is an optimal schedule. The best schedule that includes f must contain an optimal schedule for the classes that do not conflict with f, that is, an optimal schedule for L. The greedy algorithm chooses f and then, by the inductive hypothesis, computes an optimal schedule of classes from L.

The proof might be easier to understand if we unroll the induction slightly.

10) Design a Turing machine by writing the steps of the algorithm (no diagram asked) executed to accept if the input is in the language A and reject otherwise

a) A = {w#w | w ∈ {0,1}* }

1. Let M1 be a Turing machine that tests if an input string is in the language B, where B = {w#w|w ∈ {0, 1}∗ }.

2. M1 zig-zags across the tape: if no # is found, reject. Cross off symbols as they are checked to keep track.

3. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbol remained, reject, otherwise accept

b) A = {0^(2^n) | n ≥ 0}

1. Sweep left to right across the tape crossing off every other,

2. If in stage 1 tape contained a single, accept

3. If in stage 1 tape contained more that a single and the number of s was odd, reject

4. Return the head to the left-hand of the tape

5. Go to stage

c) A = {0^n(1^n) | n > 0}

```
while there are unmarked 0s, do
    Mark the left most 0
    Scan right till the leftmost unmarked 1;
    - if there is no such 1 then crash
    Mark the leftmost 1
done
Check to see that there are no unmarked 1s;
    if there are then crash
Accept
```

d) A = {0^n(1^n)(2^n) | n > 0}

```
while there are unmarked 0s, do
    Mark the left most 0
    Scan right to reach the leftmost unmarked 1;
    - if there is no such 1 then crash
Mark the leftmost 1
    Scan right to reach the leftmost unmarked 2;
    - if there is no such 2 then crash
Mark the leftmost 2
done
Check to see that there are no unmarked 1s or 2s;
    - if there are then crash
accept
```

11) Besides these, I may give you a Turing machine with its diagram and an input, where you are supposed to list all configurations showing the executions of the Turing machine and the final state (accept or reject). (See Sipser's book, exercises 3.1 and 3.2)

12) Show that if the language L and its complement L’ are recognizable, then L is decidable. (Lecture Notes)

Program P for deciding L, given programs PL and PL’ for recognizing L and L:

```
On input x
for i = 1, 2, 3, . . . (simulate in parallel PL & PL’)
    simulate PL on input x for i steps
    simulate PL’ on input x for i steps
    if either simulation accepts, break
if PL accepted, accept x (and halt)
if PL’ accepted, reject x (and halt)
```

13) Show that Ld is not recursively enumerable (recognizable). (Lecture Notes)

Recall that,

Inputs are strings over {0, 1}

Every Turing Machine can be described by a binary string and every binary string can be viewed as Turing Machine

In what follows, we will denote the ith binary string (in lexicographic order) as the number i. Thus, we can say j ∈ L(i), which means that the Turing machine corresponding to *i*th binary string accepts the *j*th binary string.

14) Show that ATM is not a decidable language. (pg.174 in Sipser’s book/ Lecture Notes)

We have already seen that Atm is r.e. Suppose (for contradiction) Atm is decidable. Then there is a TM M that always halts and L(M) = Atm. Consider a TM D as follows:

```
On input i
    Run M on input (i, i)
    Output “yes” if i rejects i
    Output “no” if i accepts i
```

Observe that L(D) = Ld! But, Ld is not r.e. which gives us the contradiction.

15) What is the ’Halting problem’? (pg.174 in Sipser’s book)

Given a program/algorithm will ever halt or not?

Halting means that the program on certain input will accept it and halt or reject it and halt and it would never goes into an infinite loop. Basically halting means terminating.

So can we have an algorithm that will tell that the given program will halt or not. In terms of Turing machine, will it terminate when run on some machine with some particular given input string.

16) Describe the (universal) TM that recognizes ATM. (Lecture Notes)

Program U for recognizing Atm:

```
On input <M,w>
    simulate M on w
    if simulated M accepts w, then accept
    else reject (by moving to q(reject))
```

U (the Universal TM) accepts <M,w> iff M accepts w. i.e.,

L(U) = Atm

But U does not decide Atm: If M rejects w by not halting, U rejects <M,w> by not halting. Indeed (as we shall see) no TM decides Atm.

17) Let L be an undecidable language. Is it possible that both L and L¯ (the complement of L) are recognizable. If yes, give an example. If no, give a short proof. (pg.181 in Sipser’s book)

NO: We have two directions to prove.

First, if L is decidable, we can easily see that both L and its complement L’ are Turing-recognizable. Any decidable language is Turing-recognizable, and the complement of a decidable language also is decidable.

For other direction, if both L and L’ are Turing-recognizable, we let M1 be the recognizer for L and M2 be the recognizer for L’. The following Turing machine M is a decider for L.

```
M =” on input string w:
    Run both M1 and M2 in parallel for the input w.
    If M1 accept, then M accept; if M2 accept M reject.
```

Thus this Turing machine M halts on every input “w” hence it is a decider. Therefore L is Turing Decidable.

18) a) What is the maximum independent set problem? (Lecture 7)

Given undirected graph G = (V , E) a subset of nodes S ⊆ V is an independent set (also called a stable set) if for there are no edges between nodes in S. That is, if u, v ∈ S then (u, v) 6∈ E.

b) What is the decision problem equivalent to this problem?

Input Graph G = (V , E) and integer k written as < G, k > Question Is there an independent set in G of size at least k?

The answer to < G, k > is YES if G has an independent set of size at least k. Otherwise the answer is NO. Sometimes we say < G, k > is a YES instance or a NO instance.

The language associated with this decision problem is L(MIS) = {< G, k >| G has an independent set of size ≥ k}

19) Show that the maximum independent set problem is in NP. (Lecture 7)

L(MIS) = {< G, k >| G has an independent set of size ≥ k} A non-deterministic polynomial-time algorithm for LMIS.

Input: < G, k > encoding graph G = (V , E) and integer k

Non-deterministically guess a subset S ⊆ V of vertices

Verify (deterministically) that

S forms an independent set in G by checking that there is no edge in E between any pair of vertices in S

|S| ≥ k.

If S passes the above two tests output YES Else NO

Key points:

string encoding S, < S > has length polynomial in length of input < G, k >

verification of guess is easily seen to be polynomial in length of < S > and < G, k >

20) Show that the minimum vertex cover problem is in NP. (Lecture 7)

LVC = {< G, k >| G has a vertex cover size ≤ k} A non-deterministic polynomial-time algorithm for LVC .

Input: < G, k > encoding graph G = (V , E) and integer k

Certificate: a subset S ⊆ V of vertices

Verifier: check the following.

S forms a vertex cover in G by checking that for each edge (u, v) ∈ E at least one of u, v is in S

|S| ≤ k.

If S passes the above two tests output YES Else NO

Key points:

certificate < S > has length polynomial in length of input < G, k >

verification of certificate easily seen to be polynomial in length of < S > and < G, k >

21) a) What is the minimum vertex cover problem? (Lecture 7)

A vertex cover of a graph is a set S of nodes such that every edge has at least one endpoint in S. In other words, we try to “cover” each of the edges by choosing at least one of its vertices.

b) What is the decision problem equivalent to this problem?

Given G and k does G have a vertex cover of size at most k? (...)

22) Show that the independent set problem can be reduced to the clique problem in polynomial time? How big is this polynomial in the worst-case? https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/clique_to_independentSet.html

For a given graph G = {V ,E} and integer k, the Clique problem is to find whether G contains a clique of size >= k.

For a given graph G’ = {V’ ,E’} and integer k’, the Independent Set problem is to find whether G’ contains an Independent Set of size >= k’.

To reduce a Clique Problem to an Independent Set problem for a given graph G = {V ,E}, construct a complimentary graph G’ = {V’ ,E’} such that

1. V = V’, that is the compliment graph will have the same vertices as the original graph.

2. E’ is the compliment of E that is G’ has all the edges that is not present in G.

Note: Construction of the complimentary graph can be done in polynomial time.

1. If there is an independent set of size k in the complement graph G’, it implies no two vertices share an edge in G’ which further implies all of those vertices share an edge with all others in G forming a clique. that is there exists a clique of size k in G.

2. If there is a clique of size h in the graph G, it implies all vertices share an edge with all others in G which further implies no two of these vertices share an edge in G’ forming an Independent Set. that is there exists an independent set of size k in G’.

23) Show that the independent set problem can be reduced to the vertex cover problem in polynomial time? How big is this polynomial in the worst-case? https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/independentSet_to_vertexCover.html

For a given graph G = {V ,E} and integer k, the Independent Set problem is to find whether G contains an Independent Set of size >= k.

For a given graph G = {V ,E} and integer k, the Vertex Cover problem is to find whether G contains a Vertex Cover of size <= k.

In a graph G = {V ,E}, “S” is an Independent Set ⇔ (V − S) is a Vertex Cover.

1. If “S” is an Independent Set, there is no edge e = (u, v) in G, such that both (u, v) ∈ S.

Hence for any edge e = (u, v), at least one of u, v must lie in (V − S)

⇒ is a vertex cover in G.

2. If (V − S) is a Vertex Cover, between any pair of vertices (u, v) ∈ S, if there exist an edge e, none of the endpoints of e would exist (V − S) in violating the definition of vertex cover.

Hence no pair of vertices in can be connected by an edge

⇒ S is an Independent Set in G.

Hence G contains an Independent Set of size k ⇔ G contains a Vertex Cover of size |V| - k.

24) Write a polynomial-time reduction from the 3SAT problem to the independent set problem. Explain in detail why the reduction works.

1. Gφ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
4. Take k to be the number of clauses

25) Write what an NP-complete problem means.

NP-complete problems are the hardest problems in NP set.

A decision problem L is NP-complete if:

1. L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
2. Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

26) State the Cook-Levin theorem. What is the importance of this theorem?

...

27) Give three examples to NP-complete problems and define each of them.

NP-complete describes the set of decision problems in NP which are NP-hard;

1. Minesweeper puzzles,
2. The set of n^2-by-n^2 Sudoku puzzles,
3. Independent Set,
4. Vertex Cover,
5. Clique,
6. Hamilton Cycle.

28) If you were to prove a problem X is NP-complete, give a proof idea by writing the two steps that one needs to prove.

To show that a problem is NP-complete, we need to show that it's both NP-hard, and in NP.

To show that it's in NP, we just need to give an efficient algorithm, which is allowed to use nondeterminism, i.e., making guesses. To show that it's NP-hard, there's a hard way, and an easy way.

The Hard way is to show that if you can solve your problem efficiently, then you can determine whether a nondeterministic Turing machine accepts its input within polynomial time.

-Fortunately, that's already been done for us, through the Cook-Levin theorem. They proved that you can encode the actions of a Turing machine with a logical formula, where you can assign values to make the formula true if and only if there is some path such that the Turing machine accepts.

-This is known as the satisfiability problem, usually abbreviated as SAT, and the theorem implies (by definition) that it's NP-hard. It's also in NP, because you can simply guess the right values for each variable, and check that it works. Together, that implies it's NP-complete.

The Easier method to produce others: simply reduce a known NP-complete problem to your problem.

-That is, prove that if you could solve your problem efficiently, you could use it to provide an efficient solution for SAT. Then you could take that solution for SAT and encode a Turing machine, etc.

-Once you've found a second NP-complete problem in this way, it can become a new source of reductions itself. And so on...

-When SAT was shown to be NP-complete, it may have been considered simply an oddity. But soon, Karp produced a new list of 21 NP-complete problems, using this new technique of reduction. Several of those reductions are taught in practically every undergraduate course on the topic.

-By now there are literally thousands of known NP-complete problems, across every area of computer science. That provides strong evidence that the class NP really is capturing an important class of problems in the world.

29) Write what an NP-hard problem means and write the difference between an NP-complete and an NP-hard problem.

A problem is considered COMPLETE for the class A if:

- It is in that class A (1);

- All problems in class A can be reduced to it (2).

A problem is considered HARD for the class A if:

- All problems in class A can be reduced to it; (2).

If someone says a problem is in NP (1) and in NP-hard (2) he is saying that it is NP-Complete.

30) Prove only one part of the proposition in Lecture 8: "If X is an NP-complete problem and someone found a polynomial-time solution for it, then that person showed P=NP (and guaranteed to get a Turing award, but no need to prove that).

Suppose X can be solved in polynomial time

1. Let $Y \in NP$. We know $Y \leq_p X$.
2. We showed that if $Y \leq_P X$ and X can be solved in polynomial time, then Y can be solved in polynomial time.
3. Thus, every problem $Y \in NP$ is such that $Y \in P$; $NP \subseteq P$.
4. Since $P \subseteq NP$, we have $P = NP$.

Since P = NP, and $X \in NP$, we have a polynomial time algorithm for X.

31) Prove only one part of the proposition in Lecture 8: "If P=NP and X is an NP-complete problem, then there is a polynomial-time solution for X."

...

32) Find a maximum flow in the given network. Write the edge-based definition of this flow.

A flow in a network G = (V, E), is a function $f : E \rightarrow R\geq0$ such that

Capacity Constraint: For each edge e, $f(e) \leq c(e)$

Conservation Constraint: For each vertex v != (s,t)

Sigma[e into v] f(e) = Sigma[e out of v] f(e)

34) Define a flow on a network.

Given a directed graph G = (V, E). Where each edge 'e' is associated with its capacity $c(e) > 0$. Two special nodes source (s) and sink (t) are given (s != t).

For every node v != (s, t), incoming flow is equal to outgoing flow.

33) Find a maximum flow in the given network. Write the path-based definition of this flow.

P: set of all paths from s to t. A flow in a network G = (V, E), is a function $f : P \rightarrow R\geq0$ such that.

Capacity Constraint: For each edge e, total flow on e is $\leq c(e)$.

{Sigma[$p\in P_e$] f(p)} $\leq c(e)$

Conservation Constraint: No need! Automatic.

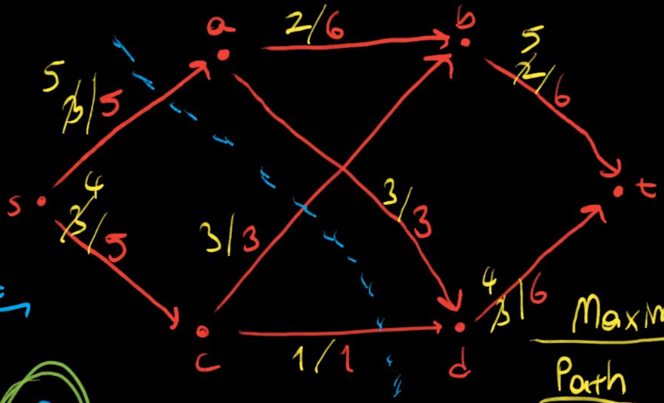
Value of flow: Sigma[$p\in P$] f (p)

35) Show the steps of the Ford-Fulkerson algorithm to find a maximum flow in the given network.

```
Set f_total = 0
Repeat until there is no path from 's' to 't':
    Run DFS from 's' to find a flow path to 't'
    Let 'f' be the minimum capacity value on the path
    Add f to f_total
    For each edge u → v on the path:
        Decrease c(u → v) by f,    Increase c(v → u) by f
```

36) Find a minimum cut in the given network. Show that it is equal to a maximum flow in this network.

Min cut problem. Delete "best" set of edges to disconnect t from s.



Minimum Cut

s-a, a-d, c-d

5 + 3 + 1 = 9

Maximum Flow

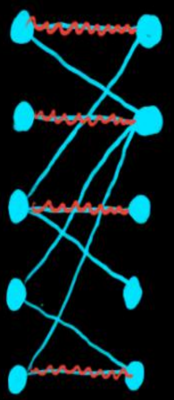
Path	Flow
s→a→d→t	3
s→a→b→t	2
s→c→b→t	3
s→c→d→t	1
+	9

Min Cut = Max Flow

37) Give a network flow problem whose solution is the same as the maximum matching problem in a bipartite graph. Explain in detail why the reduction works.

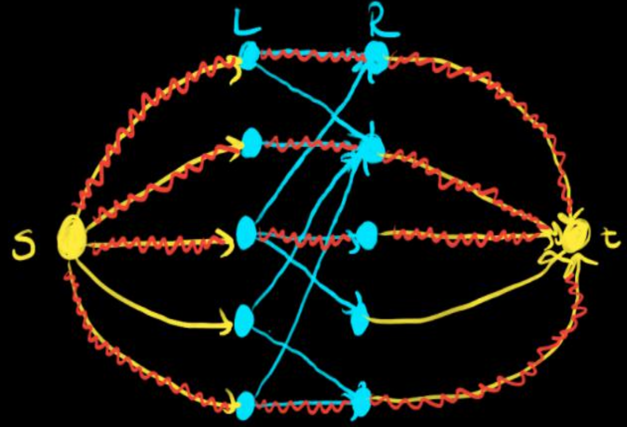
A matching in a Bipartite Graph is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matching for a given Bipartite Graph.

Bipartite Graph:



Max. Matching = 4

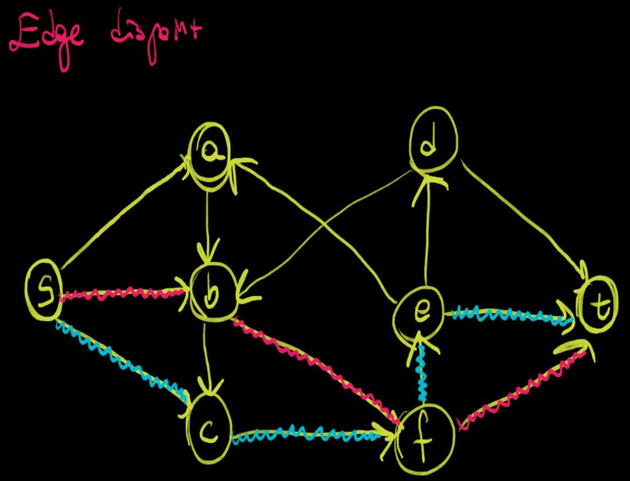
Bipartite to Flow Networks



Each capacity = 1.

38) Give a network flow problem whose solution is the same as finding the maximum number of edge-disjoint paths between two fixed vertices in a directed graph.Explain in detail why the reduction works.

Given a directed graph and two vertices in it, source 's' and destination 't', find out the maximum number of edge disjoint paths from s to t. Two paths are said edge disjoint if they don't share any edge.



This problem can be solved by reducing it to **maximum flow problem**. Following are steps.

1) Consider the given source and destination as source and sink in flow network. Assign unit capacity to each edge.

2) Run Ford-Fulkerson algorithm to find the maximum flow from source to sink.

3) The maximum flow is equal to the maximum number of edge-disjoint paths.

When we run Ford-Fulkerson, we reduce the capacity by a unit. Therefore, the edge can not be used again. So the maximum flow is equal to the maximum number of edge-disjoint paths.

39) Consider an arbitrary optimization problem. Let OPT(X) denote the value of the optimal solution for a given input X, and let A(X) denote the value of the solution computed by algorithm A given the same input X. What is required to call A a t-approximation algorithm for this problem? The answer: Under the condition that "OPT(X)/A(X)≤t and A(X)/OPT(X)≤t" for all inputs X.

…

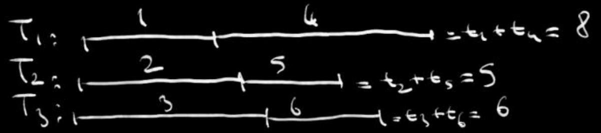
40) Define the load balancing problem.

Load Balancing

Jobs	1	2	3	4	5	6
ti	2	3	4	6	2	2

} 3 machines

Load balancing, her makineye işleri sırasıyla yerleştirmek



Make span, maximum yüklemiş makinenin yükü → $T_1 = 8$

Make span'i azaltmak için optimal çözüm uygular ve 7'ye düşürülebilir.

41) Write an approximation algorithm for the load balancing problem that gives a 2-approximation (but not lower approximation) of the optimal solution.

…

42) Write an approximation algorithm for the load balancing problem that gives a (3/2)-approximation of the optimal solution.

…

43) Example About Simplex Method >>> https://www.zweigmedia.com/RealWorld/tutorialsf4/framesSimplex.html