Homework 1 for Design and Analysis of Algorithms (Spring 2021)

Due: March 12th, 2021

Problem 1. **(Exercise 0.1 in [DPV08])** In each of the following situations, indicate whether $f = O(g)$, or $f = \Omega(g)$, or both (in which case $f = \Theta(g)$).

	$f(n)$	$g(n)$
(a)	$n - 100$	$n - 200$
(b)	$n^{1/2}$	$n^{2/3}$
(c)	$100n + \log n$	$n + (\log n)^2$
(d)	$n \log n$	$10n \log 10n$
(e)	$\log 2n$	$\log 3n$
(f)	$10 \log n$	$\log(n^2)$
(g)	$n^{1.01}$	$n \log^2 n$
(h)	$n^2/\log n$	$n(\log n)^2$
(i)	$n^{0.1}$	$(\log n)^{10}$
(j)	$(\log n)^{\log n}$	$n/\log n$
(k)	$\sqrt{n}$	$(\log n)^3$
(l)	$n^{1/2}$	$5^{\log_2 n}$
(m)	$n2^n$	3^n
(n)	2^n	2^{n+1}
(o)	$n!$	2^n
(p)	$(\log n)^{\log n}$	$2^{(\log_2 n)^2}$
(q)	$\sum_{i=1}^{n} i^k$	n^{k+1}.

Solution.

(a) $f = \Theta(g)$

(b) $f = O(g)$

(c) $f = \Theta(g)$

(d) $f = \Theta(g)$

(e) $f = \Theta(g)$

(f) $f = \Theta(g)$

(g) $f = \Omega(g)$

(h) $f = \Omega(g)$

(i) $f = \Omega(g)$

(j) $f = \Omega(g)$

$$\begin{aligned}\log n &= \log_{\log n} n * \log\log n \\ f(n) &= (\log n)^{\log_{\log n} n * \log\log n} \\ &= n^{\log\log n} \\ &= \Omega(g(n))\end{aligned}$$

(k) $f = \Omega(g)$

(l) $f = O(g)$

$$\begin{aligned}g(n) = 5^{\log_2 n} &= 2^{\log_2 5 \log_2 n} \\ &= n^{\log_2 5} = \Omega(f(n)) \\ f &= O(g)\end{aligned} \tag{1}$$

(m) $f = O(g)$

(n) $f = \Theta(g)$

(o) $f = \Omega(g)$

(p) $f = O(g)$

(q) k>-1 $f = \Theta(g)$, k=-1 $f = \Omega(g)$.

1 k>0

$$\begin{aligned}f(n) &< \int_1^{n+1} x^k dx = \Theta(n^{k+1}) \\ f(n) &> \int_0^{n} x^k dx = \Theta(n^{k+1}) \\ f(n) &= \Theta(g(n))\end{aligned}$$

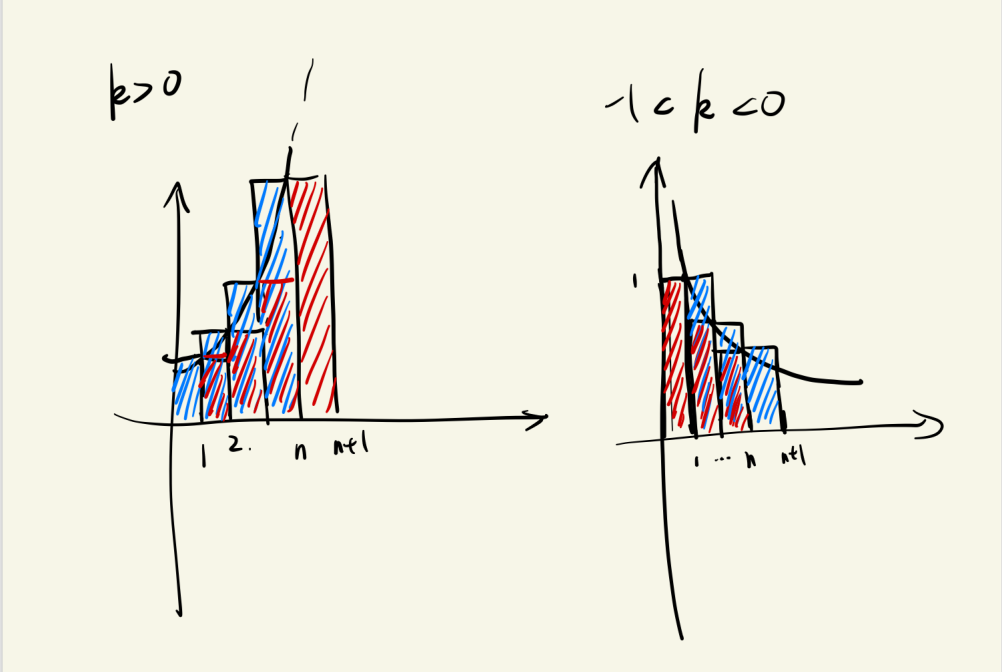
2 k=0 Obviously, $f(n) = \Theta(g(n))$

3 -1<k<0

$$\begin{aligned}f(n) &< 1 + \int_1^{n} x^k dx = \Theta(n^{k+1}) \\ f(n) &> \int_1^{n+1} x^k dx = \Theta(n^{k+1}) \\ f(n) &= \Theta(g(n))\end{aligned}$$

4 k=-1 $f(n) = \Theta(\log n) = \Omega(g(n))$

5 k<-1 Both $f(n)$ and $g(n)$ are decreasing.



□

Problem 2. (Exercise 0.2 in [DPV08]) Show that, if c is a positive real number, then $g(n) = 1+c+c^2+\cdots+c^n$ is:

(a) $\Theta(1)$ if $c < 1$.

(b) $\Theta(n)$ if $c = 1$.

(c) $\Theta(c^n)$ if $c > 1$.

Problem 3. Show that there exists a C++ program P who can generate all pairs of natural numbers (x, y) such that $P_x(y)$ terminates [1]. That is, the program P can keep printing pairs of numbers (x, y) such that $P_x(y)$ terminates and for every (x', y') such that $P_{x'}(y')$ terminates, the program P can print it at some time.

Solution. □

[1] Since the number of such pairs are infinite, the program P must run forever.

Algorithm 1: Solution 3

```
1  F ← ∅// pairs have been output
2  for S = 1 : +∞ do
3      for T = 1 : S do
4          for x = 0 : S − T do
5              y = S − T − x
6              Simulate P_x(y) for T steps// (or seconds, hours, . . . )
7              if P_x(y) terminates in T steps then
8                  if (x, y) ∉ F then
9                      Output (x, y)
10                     F ← F ∪ {(x, y)}
11                 end
12             end
13         end
14     end
15 end
```

References

[DPV08] Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Virkumar Vazirani. *Algorithms*. McGraw-Hill Higher Education New York, 2008. 1, 3

Algorithm Design and Analysis
Assignment 2

1. You are given two sorted lists of size m and n. Give an $O(\log m + \log n)$ time algorithm for computing the k-th smallest element in the union of the two lists.

   Solution. Assume that the two lists are A and B, and $|A| + |B| = m + n \geq k$. Now we divide each lists into two parts with the same length, and compare the element in the middle. Assume they are $A[i]$ and $B[j]$ (The index start from 1 rather than 0). Now here are 4 situations:

   (a) $A[i] > B[j]$

   If $k < i + j$, the the $k-$th smallest element may not occur in $A[i :]$ because they are greater than these $i + j$ elements, so we drop$A[i :]$.

   If $k \geq i + j$, the the $k-$th smallest element may not occur in $B[: j]$ because they are smaller than $A[i]$, we drop them and change k to $k - j$.

   (b) $A[i] \leq B[j]$ is similar to the first condition.

   See algorithm1. □

2. A k-way merge operation. Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements. Design a efficient algorithm using divide-and-conquer.

   Solution. We recursively divide the sets into two parts and merge their sorted results. See algorithm2.

   We need to merge $k/2$ pairs of arrays of length n, $k/4$ pairs of length $2n$,..., and finally 1 pair of length $kn/2$. There are $\log k$ layers in total, and each layer cost $O(nk/2) = O(nk)$.

   The total complexity is $O(nk \log k)$ □

Algorithm 1: Find the k-th smallest element in two ordered lists

Input: A, B

// Assume that $|A| + |B| \geq k$

Output: *result*

```
while |A||B| > 0 and k > 1 do
    i ← ⌈|A|/2⌉
    j ← ⌈|B|/2⌉
    if A[i] > B[j] then
        if k<i+j then
            A ← A[: i − 1]
        else
            B ← B[j + 1 :]
            k ← k − j
    else
        if k<i+j then
            B ← B[: j − 1]
        else
            A ← A[i + 1 :]
            k ← k − i
end
if |A| = 0 then
    result ← B[k]
else if |B| = 0 then
    result ← A[k]
else if r = 1 then
    result ← Min{A[1], B[1]}
```

Algorithm 2: MergeLists

Input: A set of sorted arrays A

Output: A sorted array $result$

```
if |A| = 0 then
    result ← [ ]
else if |A| = 1 then
    result ← A[1]
else
    i = ⌈|A|/2⌉
    L ← MergeLists(A[: i])
    R ← MergeLists(A[i + 1 :])
    result ← MergeSort(L, R)
```

3. Recall that we have learned how to find the k-th element in a list with a randomized algorithm (randomly choose a pivot), can we do it deterministically? In this exercise, we will develop one called the *Median of Medians* algorithm, invented by Blum, Floyd, Pratt, Rivest, and Tarjan.

   Why we need to pick the *pivot* x randomly in our randomized algorithm? This is to guarantee, at least in expectation, that the numbers less than x and the numbers greater than x are in close proportion. In fact, this task is quite similar to the task of "finding the k-th largest number" itself, and therefore we can bootstrap and solve it recursively!

   Assume we have an array A of n distinct numbers and would like to find its k-th largest number.

   (a) Consider that we line up elements in groups of three and find the median of each group. Let x be the median of these $n/3$ medians. Show that x is close to the median of A, in the sense that a constant fraction of numbers in a is less than x and a constant fraction of numbers is greater than x as well.

   Solution. $1/3$ □

   (b) Design a recursive algorithm by the above idea and analyze the running time.

   Solution. Reference: [Wik21]

   We use *Median of Medians* to find a proper *pivot* x to recursively find the $k - th$ element. Also, we need to use this algorithm itself to find the median of medians, which implies another recursion.

   By previous analysis, this reduce the size to at least $1/3$. Here are two recursions, a $T(n/3)$ for computing median of the $n/3$ medians, and $T(\gamma n)$ for computing the

Algorithm 3: QuickSelect(A, k)

Input: List A, nubmer k

Output: *result*

```
B ← Medians(A, 3)
X ← QuickSelect(B, ⌈|B|/2⌉)
// Recursively call QuickSelect to get the median of medians beacuse
   find median is a special case of find the k-th smallest element.
// Choose X as pivot
i ← Partition(A, X)// Partition A by pivot X, i is the index of X in A
   after partition
if i>k then
    result ← QuickSelec(A[: i − 1], k)
else if i<k then
    result ← QuickSelec(A[i + 1 :], k − i)
else
    result ← X
```

remaining list where γ is between $1/3$ and $1 - 1/3$. We have

$$2T(\frac{n}{3}) + cn \leq T(n) \leq T(\frac{n}{3}) + T(\frac{2n}{3}) + cn$$

for some constant c.

The worst case will be $T(n) = O(n \log n)$.

(*Hint:* It is not proper to use *Median of Medians* recursively to get the *pivot*. The *pivot* may be far from the true median after several recursions, and you cannot use the $1/3$ constant because the *pivot* you get this way is the median of medians of …of medians rather than the median of medians). □

(c) Can we improve the running time by increasing the number of elements (e.g. 4,5,6?) in each group? What is the best choice? Give the answer and the proof. Note that in this problem, we drop the big-O notation and discuss about the constants.

Solution. Reference: [Wik21]

We only consider odd numbers here because it's quicker to compute the median.

Assume that there are g elements in each group.

Similarly, we get

$$\begin{aligned} T(n) &\leq T(\frac{n}{g}) + T((1 - \frac{1}{2} \cdot \frac{1}{g} \cdot \frac{g+1}{2})n) + cn \\ &= T(\frac{n}{g}) + T(\frac{3g-1}{4g}n) + cn \end{aligned}$$

for the worst case.

$T(n) = O(n)$ can be proved inductively. Now we consider the constant. Assume that $T(n) = \alpha n$, we get

$$\begin{aligned} \alpha n &= \frac{\alpha n}{g} + \frac{3g-1}{4g}\alpha n + cn \\ \alpha &= \frac{4g}{g-3}c \\ &= (4 + \frac{12}{g-3})c \end{aligned}$$

It seems that the constant factor decreases as g increases. However, the constant c increases as g increases.

The constant c is about the cost of finding the median in each group of size g. If we choose insertion sort to find the median, we cost $g(g-1)/2$ comparisons in each group and n comparisons for sorting under the worst condition. Now,

$$\begin{aligned} cn &= \frac{n}{g} \cdot \frac{g(g-1)}{2} + n \\ c &= \frac{g+1}{2} \end{aligned}$$

Then

$$\alpha = (4 + \frac{12}{g-3})\frac{g+1}{2}$$

When $g = 7$ (Or 5. The answer depends on your analysis about the constant c) we get the minimal α. □

4. The Hadamard matrices $H_0,\ H_1,\ H_2, ...$ are defined as follows:

   - H_0 is the 1×1 matrix [1].
   - For $k > 0$, H_k is the $2^k \times 2^k$ matrix

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

   Show that if $\vec{v}$ is a column vector of length $n = 2^k$, then the matrix-vector product $H_k\vec{v}$ can be calculated using $O(n \log n)$ operations in word RAM.

   Solution. Let $\vec{v} = \begin{pmatrix} \vec{v}_1 \\ \vec{v}_2 \end{pmatrix}$ where $\vec{v}_1$ and $\vec{v}_2$ are column vectors of length 2^{k-1}

$$H_k\vec{v} = \begin{pmatrix} H_{k-1}(\vec{v}_1 + \vec{v}_2) \\ H_{k-1}(\vec{v}_1 - \vec{v}_2) \end{pmatrix}$$

   This product can be calculated recursively. Notice that $\vec{v}_1, \vec{v}_2,\ \vec{v}_1 + \vec{v}_2$ and $\vec{v}_1 - \vec{v}_2$ can be computed in $O(n)$ time.

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= O(n \log n) \end{aligned}$$

   □

5. How long does it take you to finish the assignment (include thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty.

References

[Wik21] Wikipedia contributors. Median of medians — Wikipedia, the free encyclopedia, 2021. [Online; accessed 12-April-2021].

Algorithm Design and Analysis
Assignment 3

1. The following algorithm attempts to find the shortest path from node s to node t in a directed graph with some negative edges:

   - Add a large enough number to each edge weight so that all the weights become positive, then run Dijkstra's algorithm.

   Either prove this algorithm correct, or give a counter-example.

   Solution. It is not correct. Assume the number is c, the orginal weight of a path is w, then the new weight becomes $w' = w + kc$, where k is the length of the path.

   In a word, different paths get different additional weights.

   A counter-example: there are three vertices a, b, and c, where $w(a, b) = w(a, c) = 2, w(b, c) = -1$. The shortest path from a to c is $(a, b), (b, c)$ with weight 1, but it is replaced by (a, c) with weight $2 + c$ for any additional integer $c > 1$ in the modified graph. □

2. A bipartite graph is a graph $G = (V, E)$ whose vertices can be partitioned into two sets V_1 and V_2 (i.e., $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$) such that there are no edges between vertices in the same set. (For instance, if $u, v \in V_2$, then there is no edge between u and v.)

   (a) Show that an undirected graph is bipartite if and only if it contains no cycles of odd length.

   (b) Give a linear-time algorithm to determine whether an undirected graph is bipartite.

   Solution. (b) DFS to color the vertices with two colors, where adjacent vertices must be colored differently. If conflict occurs, then this graph is not a bipartite graph. □

3. Let $G = (V, E)$ be an undirected connected graph. Let T be a depth-first search tree of G. Suppose that we orient the edges of G as follows: For each tree edge, the direction is from the parent to the child; for every non-tree (back) edge, the direction is from the descendent to the ancestor. Let G' denote the resulting directed graph.

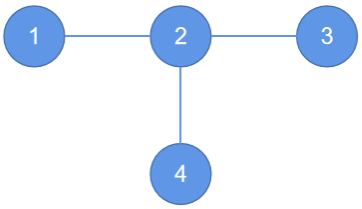
   (a) Give an example to show that G' is not strongly connected.

   (b) Prove that if G' is strongly connected, then G satisfies the property that removing any single edge from G will still give a connected graph.

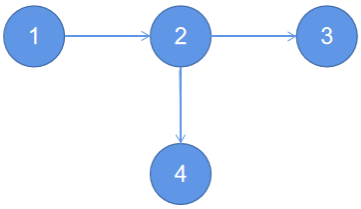
   (c) Prove that if G satisfies the property that removing any single edge from G will still give a connected graph, then G' must be strongly connected

   (d) Give an efficient algorithm to find all edges in a given undirected graph such that removing any one of them will make the graph no longer connected.

Solution.



G



G'

(a)

(b) Suppose that we remove (i, j) from G, and $(i, j) \in G', (j, i) \notin G'$ without loss of generality. If G' is strongly connected, there exists some path p from j to i in G'. So, the corresponding path in $G \backslash (i, j)$ makes i and j still connected, which implies that $G \backslash (i, j)$ is still connected.

(c) If we prove that both paths form i to j and j to i exists for arbitrary $(i, j) \in G$, G' is strongly connected.

Suppose that we remove (i, j) from G, and $(i, j) \in G', (j, i) \notin G'$ without loss of generality. If $G \backslash (i, j)$ is still connected, there exists some path p connect i and j in G. Assume it is $p_0, p_1, \ldots, p_k$ with length k where $p_0 = j$ and $p_k = i$.

Now prove that there exists some path p' form j to i in G'.

If the direction of (p_0, p_1) in G' is $j \to p_1$, then p_1 is a child node of j in the DFS tree, we have $(j, p_1) \in G'$. Otherwise, both i and p_1 pointing to j implies that p_1, i is a descendent of j.

1. If i is the descendent, (i, j) is the back edge, there is still a path form j(ancestor) to i(descendent). Now we get the proof.
2. If p_1 is the descendent, we replace (p_1, j) by a path from j to p_1 and denote it by $[j \to p_1]$

Whatever the direction of (p_1, j) is, we get a path from j to p_1. Applying this method on $(p_1, p_2), \ldots, (p_{k-1}, i)$, we get the path form j to i.

Since $G \backslash (i, j)$ is connected for any$(i, j) \in G$, $\forall p \neq q \in G$ there exists a path connecting p, q. For any edge in this path, the two vertices are reachable from each other, so p, q is also reachable from each other. G' is strongly connected.

Hint: Notice that a path in an undirected graph does not represent the same path in the corresponding directed graph, and a cut edge in a directed graph does not correspond to a cut edge in the undirected graph, either (Although they may hold in some special cases, you need to describe your proof in detail).

(d) Tarjan algorithm to find all of the cut edges.

□

4. How long does it take you to finish the assignment (include thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty.

Algorithm Design and Analysis
Assignment 4

1. You are given a set of n jobs, where each job j is associated with a size s_j (how much time it takes to process the job) and a weight w_j (how important the job is). Suppose you have only one machine that can process one unit of jobs per time slot. Assume all jobs are given at time $t_0 = 0$ and are to be processed one by one using this machine. Let $C_j > t_0$ be the time that job j is completed. The goal is to find a schedule (of all the jobs) that minimizes the weighted completion time, i.e.,$\sum_{j=1}^{n} w_j C_j$.

   Greedy algorithms first order the jobs according to some criteria, and then process them one by one. Only one of the following criteria gives an algorithm that minimizes the weighted completion time.

   - Highest Weight First: process jobs in in descending order of their weights.
   - Smallest Size First: process jobs in ascending order of their sizes.
   - Highest Density First: process jobs in descending order of their weight to size ratio, namely in descending order of w_j/s_j.

   (a) Find out which one is correct.

   (b) Reason about its correctness.

   (c) Give counter examples for the other criteria.

   Solution.

   (a) Highest Density First is correct.

   (b) Given jobs i,j with $w_i/s_i > w_j/s_j$, we say that (i, j) is an **inversion** in schedule S, if S schedules j before i.

   Observations:

   1. The greedy algorithm does not have idle time
   2. The optimal algorithm does not have idle time.
   3. according to the greedy algorithm, the number of inversion is 0.
   4. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

   Then we show that swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the weighted completion time.

   Proof. : suppose that (i,j) is an inversion in S with $\frac{w_i}{s_i} > \frac{w_j}{s_j}$, and S schedule j before i, and j starts at time T, after the swapping ($S^{'}$), the difference between the weight completion time between S and $S^{'}$ is

$W(S) - W(S') = w_j(T + s_j) + w_i(T + s_j + s_i) - (w_i(T + s_i) + w_j(T + s_i + s_j)) = w_i s_j - w_j s_i > 0$, where the last inequality is due to $\frac{w_i}{s_i} > \frac{w_j}{s_j}$.
So repeating this process shows the optimality of greedy. □

(c) 1. Counter example for Highest Weight First:
Suppose there are two jobs, job 1 with $s_1 = 100, w_1 = 10$, and job 2 with $s_2 = 10, w_2 = 9$. According to the greedy algorithm with criterion highest weight first, job 1 is before job 2, the weighted completion time is

$$W = 10 * 100 + 9 * (100 + 10).$$

And the optimal algorithm is to schedule job 2 before job 1, and the weight completion time is

$$W' = 9 * 10 + 10 * (10 + 100).$$

It's easy to see $W > W'$.

2. counter example for Smallest Size First:
Suppose there are two jobs: job 1, with $s_1 = 10, w_1 = 1$, and job 2, with $s_2 = 20, w_1 = 100$.
According to the greedy algorithm with criterion smallest size first, job 1 is before job 2, the weighted completion time is

$$W = 1 * 10 + 100 * (10 + 20).$$

And the optimal algorithm is to schedule job 2 before job 1, and the weight completion time is

$$W' = 100 * 20 + 1 * (10 + 20).$$

It's easy to see $W > W'$.

□

2. Let $G = (V, E)$ be a connected undirected graph with positive edge weights. Give an algorithm to find a subset of edges E' with the smallest total weight such that removing E' from G will leave a graph with no cycle. Note that E' must contain at least one edge on every cycle of G. You need to prove the correctness of your algorithm.

Solution. Let $E^* = E - E'$.

Given E, the task of finding E' with the smallest total weight is equivalent to finding E^* with largest total weight where we require $G^* = (V, E^*)$ does not contain any cycle.

Notice that $G^* = (V, E^*)$ must be connected since otherwise we can always add edge into E^* making its total weight larger while not breaking the no-cycle constraint.

Since G^* is acyclic and connected, G^* is a tree. Thus, finding G^* with E^* of largest total weight is equivalent to finding the Maximum Spanning Tree of the original graph G. To find the maximum spanning tree, we can first apply the following transformation to all edge weights:

$$w^*_{\mathrm{u}} = -w_{\mathrm{u}} + MAX_{v:v\in E}\{w_v\} \quad \forall u \in E$$

After such transformation, all edge weights are still non-negative. Hence, we can run Prim's algorithm or Kruskal's algorithm to find the Minimum Spanning Tree of the graph under the transformed edge weights $\{w^*_u\}$.

Firstly, adding a constant number to all edge weights does not change the MST of a graph. Secondly, reversing the signs of all edge weights makes the Minimum Spanning Tree found be the actual desired Maximum Spanning Tree under the graph with pre-transformation edge weights $\{w_u\}$.

After we have found the Maximum Spanning Tree E^* of G, we return $E' = E - E^*$ as final answer. Overall runtime is $O(E \log V)$ □

3. Alice wants to throw a party and is deciding whom to call. She has n people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know and five other people whom they don't know. Give an efficient algorithm that takes as input the list of n people and the list of pairs who know each other and outputs the best choice of party invitees. Give the running time in terms of n.

Solution.

(a) Algorithm:

- Construct an undirected graph $G(V, E)$ for these people(adjacent matrix), where each vertex represents a person and each $(i, j) \in E$ implies that i and j know each other.
- We recursively find and delete a vertex with degree d where $d < 5$ or $|V| - d - 1 > 5$ until there is no such vertex.
- The set of vertices left is exactly the best choice.

(b) Optimality:

If a vertex with degree d satisfies $5 \leq d \leq |V| - 6$, we call it a "good" vertex, otherwise it is "bad".

Notice that deleting a vertex is likely to transform a remaining "good" vertex to a "bad" one but will never transform a "bad" vertex to a "good" one because the deletion reduces either d or $|V| - d - 1$.

Assume there is a optimal algorithm OPT, the deletion of which in i-th round is d_i. The set of remaining vertices is R and the set of deleted vertices is D. $|R| \cap |D| = \emptyset$, $|R| \cup |D| = |V|$.

Similarly, in our algorithm, call the deletion in i-th round d'_i. The set of remaining vertices R' and the set of deleted vertices D'.

Now it is easy to prove that $D' \subseteq D$ by induction:

- In the first round, d'_i is deleted, which implies it is "bad". So it will be deleted in OPT too (though may not in the first round) because it will never change to "good".
- In the i-th round ($i > 1$), we have that $d'_1, \ldots, d'_{i-1}$ are deleted, and now d'_i is "bad" too (though it may be "good" before). Because $d'_1, \ldots, d'_{i-1}$ will be deleted in OPT too, d'_i will not be "better" in OPT than in our algorithm. d'_i will also be deleted in OPT.
- Finally, we have that $D' \subseteq D$, which implies $|R'| \geq |R|$. Because $|R|$ is optimal, we have $|R'| = |R|$, our algorithm is also optimal.

(c) Running time:

There are at most n deletions, so it runs at most n rounds. In each round, it takes up to $O(n)$ time each to search and to update if we store the degrees in an array. The total running time is $O(n^2)$

□

Algorithm Design and Analysis
Assignment 5

1. Consider the following 3-PARTITION problem. Given integers $a_1, ..., a_n$, we want to determine whether it is possible to partition of $\{1, ..., n\}$ into three disjoint subsets I, J, K such that

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{1}{3} \sum_{i=1}^{n} a_i.$$

For example, for input $(1, 2, 3, 4, 4, 5, 8)$ the answer is yes, because there is the partition $(1, 8),\ (4, 5),\ (2, 3, 4)$. On the other hand, for input $(2, 2, 3, 5)$ the answer is no. Devise and analyze a dynamic programming algorithm for 3-PARTITION that runs in time polynomial in n and in $\sum_i a_i$.

Solution. (Remark: all integers are positive.) Let $f(j, x, y)$ be a boolean value to show whether we can only use the first j integers to make two disjoint subsets with sum x and y. We define the transition of f to be

$$f(j, x, y) = f(j - 1, x - a_j, y) \vee f(j - 1, x, y - a_j) \vee f(j - 1, x, y).$$

It is easy to show if any one of $f(j - 1, x - a_j, y)$, $f(j - 1, x, y - a_j)$ and $f(j - 1, x, y)$ is true, then we can directly put the j-th integer into the corresponding subset, which means $f(j, x, y)$ is also true. On the other hand if $f(j, x, y)$ is true, remove a_j from the corresponding subset shows one of the three must be true. With the transition function, we give the DP algorithm as follows. □

Algorithm 1: Determine whether we can make a feasible 3-PARTITION

```
f[0,0,0] ← 1. ;
S = Σ_{i=1}^{n} a_i/3 ;
for j = 1 to n do
    for x = 1 to S do
        for y = 1 to S do
            f[j,x,y] ← f[j−1,x,y] ;
            if x ≥ a[j] and f[j−1,x−a[j],y] = 1 then
                f[j,x,y] ← 1;
            end
            if y ≥ a[j] and f[j−1,x,y−a[j] = 1 then
                f[j,x,y] ← 1;
            end
        end
    end
end
Output f[n, Σ_{i=1}^{n} a_i/3, Σ_{i=1}^{n} a_i/3] ;
```

2. Let $X[1..n]$ be a reference DNA sequence. Let S be a set of m exon candidates of X, where each exon candidate is represented by a triple (i, j, w), which means that the strength or probability for fragment $X[i..j]$ being an exon is w. Notice that many triples in S are false exons, and true exons do not overlap. Show how to use dynamic programming to find a maximum-weight subset of S in which all exon candidates are non-overlapping. The time complexity should be linear in terms of n and m.

   Solution. For any $s = (i, j, w) \in S$, we define $i(s) = i$, $j(s) = j$, and $w(s) = w$. Let $f(k)$ be the maximum weight construct by a feasible subset of exons with $j \leq k$. Let S_k be the subset of exons with $j = k$, we define the transition of f as follows:

$$f(k) = \max\{f(k-1), \max_{s \in S_k}\{f(i(s)-1) + w(s)\}\}.$$

   We can prove it by induction and $f(0) = 0$ trivially holds. Assume $f(k')$ holds for any $k' < k$, we can at most choose one exon in S_k. That means we can either choose nothing in S_k and get $f(k-1)$ or choose one $s \in S_k$ so that we can only use other exons with $j < i(s)$ and get $f(i(s)-1) + w(s)$. Hence, we can conclude the transition, and we give the DP algorithm below. □

Algorithm 2: Calculate the maximum weight subset

```
1  f[0] ← 0. ;
2  for k = 1 to n do
3  |   f[k] ← f[k − 1] ;
4  |   for each s ∈ S_k do
5  |   |   if f[i(s) − 1] + w(s) < f[k] then
6  |   |   |   f[k] ← f[i(s) − 1] + w(s) ;
7  |   |   end
8  |   end
9  end
10 Output f[n] ;
```

3. Consider the following game. A "dealer" produces a sequence $s_1, \ldots, s_n$ of "cards" facing up, where each card s_i has a value v_i. Then two players take turns picking a card from the sequence, but can only pick the first or the last card of the (remaining) sequence. The goal is to collect cards of largest total value. Assume n is even.

   (a) Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. That is, the natural *greedy* stratgy is suboptimal.

   (b) Give an $O(n^2)$ algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute in $O(n^2)$ time some information, and then the first player should be able to make each move optimally in $O(1)$ time by looking up the precomputed information.

Solution.

(a) Consider the sequence $2, 100, 1, 1$, if the first player chooses 2, then the second player can choose 100, which means the greedy strategy is not optimal.

(b) Define $f[i, j]$ $(j - i \geq 1)$ be the optimal value of the player when he is choosing cards from the sequence $s_i, ..., s_j$. (Notice that we allow there is an odd number of cards). We define the transition as follows:

If $j - i = 1$, $f[i, j] = \max\{a_i, a_j\}$.

If $j - i > 1$,

$$f[i, j] = \max\{a_i + \sum_{i+1 \leq k \leq j} a_k - f[i + 1, j], a_j + \sum_{i \leq k \leq j-1} a_k - f[i, j - 1]\}.$$

When $j-i=1$, it's easy to see the best strategy is to choose the larger card. If $j-i>1$, there are two choices for the player, choosing a_i or a_j. Assume we get the correct value of $f[i',j']$ for all $j'-i'<j-i$, when the player (player A) chooses a_i, the other player will get $f[i+1,j]$, and player A will get $a_i+\sum_{i+1\le k\le j}a_k-f[i+1,j]$. With the same reason, the player will get $a_j+\sum_{i\le k\le j-1}a_k-f[i,j-1]$ if he chooses a_j. We give the algorithm below and we record the optimal strategy by the array g. That means, when the first player is choosing card from the subsequence from i to j, he should choose i or j following $g[i,j]$.

Algorithm 3: Calculate the optimal strategy.

$\forall 1\le i\le n, f[i]\leftarrow i.$;
for $k=1$ *to* $n-1$ **do**
  for $i=1$ *to* $n-k$ **do**
    $j\leftarrow i+k$;
    if $a_i+\sum_{i+1\le k\le j}a_k-f[i+1,j]>a_j+\sum_{i\le k\le j-1}a_k-f[i,j-1]$ **then**
      $f[i,j]\leftarrow a_i+\sum_{i+1\le k\le j}a_k-f[i+1,j]$;
      $g[i,j]\leftarrow i$;
    end
    if $a_i+\sum_{i+1\le k\le j}a_k-f[i+1,j]\le a_j+\sum_{i\le k\le j-1}a_k-f[i,j-1]$ **then**
      $f[i,j]\leftarrow a_j+\sum_{i\le k\le j-1}a_k-f[i,j-1]$;
      $g[i,j]\leftarrow j$;
    end
  end
end
Output g ;

□

4. Assume points $v_1, v_2, \ldots, v_n$ form a convex polygon in $\mathbb{R}^2$. Let $d(i,j)$ be the Euclidean distance between v_i and v_j if $i \le j$ and $d(i,j) = -\infty$ if $i > j$. For every $r \ge 0$, we use $d^{(r)}(i,j)$ to denote the length of the the *longest paths* from v_i to v_j using *at most* r edges. Therefore, $d(i,j) = d^{(1)}(i,j)$.

   (a) Let $s, t \ge 0$ be any two any integers satisfying $r = s + t$. For every $i \le j$, prove that $d^{(r)}(i,j) = \max_{i \le k \le j} \left\{ d^{(s)}(i,k) + d^{(t)}(k,j) \right\}$.

   (b) Prove that the distance $d(\cdot,\cdot)$ satisfies the *inverse Quadrangle Inequality* (iQI):

   $$\forall i \le i' \le j \le j' : d(i,j) + d(i',j') \ge d(i',j) + d(i,j').$$

   (c) Prove that for any integer $r \ge 0$, $d^{(r)}(\cdot,\cdot)$ satisfies iQI as well.

   (d) If we let $K^{(r)}(i,j)$ denote $\max \left\{ k \mid i \le k \le j \text{ and } d^{(r)}(i,j) = d^{(s)}(i,k) + d^{(t)}(k,j) \right\}$, prove that

   $$K^{(r)}(i,j) \le K^{(r)}(i,j+1) \le K^{(r)}(i+1,j+1), \quad \text{for} \quad i \le j.$$

   (e) Give an algorithm to compute $d^{(r)}(i,j)$ for all $1 \le i < j \le n$ in $O(\log r \cdot n^2)$ time [1].

Solution.

(a) Assume $d^{(r)}(i,j)$ using $r' \le r$ edges $(i_1 = i, i_2), (i_2, i_3) \ldots, (i_{r'}, j)$. $\forall 1 \le k \le r'$, $i_k < j$ because otherwise there is $-\infty$ distance. For any s and t, we can partition the r' edges into two subsets by the pivot k, with s edges and $r' - s$ edges where $r' - s \le t$. Thus, it is one of the choice of $\max_{i \le k \le j} \left\{ d^{(s)}(i,k) + d^{(t)}(k,j) \right\}$.

(b) The iQL trivially holds when $i = i'$ or $j = j'$, the remaining case is 1) $i < i' = j < j'$ and 2) $i < i' < j < j'$. In case 1), the iQL becomes the triangle inequality $d(i,j) + d(j,j') \ge d(j,j')$ and it holds for the Euclidean distance. In case 2), (v_i, v_i', v_j, v_j) is a convex quadrilateral so the diagonals (v_i, v_j) and $(v_{i'}, v_{j'})$ are inside the quadrilateral and their intersection point o are also inside. Refer to Figure 1, we have $io + oj' \ge ij'$ and $i'o + oj \ge i'j$, so $d(i,j) + d(i',j') \ge d(i',j) + d(i,j')$.

[1] That is, your algorithm needs to compute all the $\binom{n}{2}$ values within $O(\log r \cdot n^2)$ time.

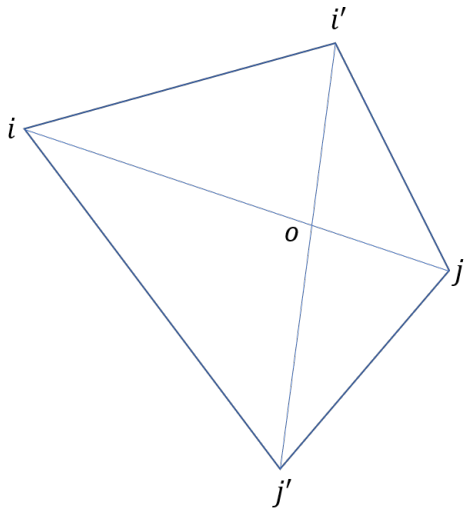


Figure 1: example

(c) We prove it by induction on r, and the base case is proved above. To prove iQI holds inductively on r, we assume it holds for any $r' < r$. Fix $s = r - 1$ and $t = 1$, we have that $d^{(r)}(i,j) = \max_{i \le k \le j} d^{(r-1)}(i,k) + d(k,j)$. Consider the two terms at the RHS of the iQL, assume $d^{(r)}(i',j)$ is maximized at $k = x$ and $d^{(r)}(i,j')$ is maximized at $k = y$. That means

$$d^{(r)}(i',j) = d^{(r-1)}(i',x) + d(x,j),\ d^{(r)}(i,j') = d^{(r-1)}(i,y) + d(y,j') \tag{1}$$

If $y \ge x$, because $i \le i' \le x \le j$ and $i \le x \le y \le j'$, we have

$$d^{(r)}(i,j) \ge d^{(r-1)}(i,x) + d(x,j),\ d^{(r)}(i',j') \ge d^{(r-1)}(i',y) + d(y,j').$$

Then, because $i \le i' \le x \le y$, by the induction hypothesis, we have

$$d^{(r-1)}(i,x) + d^{(r-1)}(i',y) \ge d^{(r-1)}(i',x) + d^{(r-1)}(i,y). \tag{2}$$

Hence,

$$\begin{aligned} d^{(r)}(i,j) + d^{(r)}(i',j') &\ge d^{(r-1)}(i,x) + d(x,j) + d^{(r-1)}(i',y) + d(y,j') \\ &\ge d^{(r-1)}(i',x) + d^{(r-1)}(i,y) + d(x,j) + d(y,j') \quad \text{By Eqn (2)} \\ &= d^{(r)}(i',j) + d^{(r)}(i,j') \quad \text{By Eqn (1)} \end{aligned}$$

If $y \le x$, symmetrically, we have

$$d^{(r)}(i,j) \ge d^{(r-1)}(i,y) + d(y,j),\ d^{(r)}(i',j') \ge d^{(r-1)}(i',x) + d(x,j').$$

Because $y \le x \le j \le j'$, by the induction hypothesis, we have

$$d(y,j) + (x,j') \ge d(x,j) + d(y,j'). \tag{3}$$

Hence,

$$\begin{aligned} d^{(r)}(i,j) + d^{(r)}(i',j') &\ge d^{(r-1)}(i,y) + d(y,j) + d^{(r-1)}(i',x) + d(x,j') \\ &\ge d^{(r-1)}(i,y) + d^{(r-1)}(i',x) + d(x,j) + d(y,j') \quad \text{By Eqn (3)} \\ &= d^{(r)}(i',j) + d^{(r)}(i,j') \quad \text{By Eqn (1)} \end{aligned}$$

(d) To prove the first inequality, we plan to show that

$$\begin{aligned} \forall i \le k < j, \quad & d^{(s)}(i,k+1) + d^{(t)}(k+1,j) - d^{(s)}(i,k) - d^{(t)}(k,j) \\ & \le d^{(s)}(i,k+1) + d^{(t)}(k+1,j+1) - d^{(s)}(i,k) - d^{(t)}(k,j+1). \end{aligned} \tag{4}$$

If it holds, then move k from i to $K^{(r)}(i,j)$, $d^{(s)}(i,k) + d^{(t)}(k,j+1)$ must increase at least as much as $d^{(s)}(i,k) + d^{(t)}(k,j)$. Thus, $K^{(r)}(i,j+1) \ge K^{(r)}(i,j)$. To prove it, we should have

$$d^{(t)}(k+1,j) - d^{(t)}(k,j) \le d^{(t)}(k+1,j+1) - d^{(t)}(k,j+1).$$

Notice that it is implied by the iQL of $d^{(t)}(\cdot,\cdot)$ for $k \le k+1 \le j \le j+1$.

Similarly, to prove the second one, we plan to show that

$$\begin{aligned} \forall i < k < j, \quad & d^{(s)}(i,k+1) + d^{(t)}(k+1,j+1) - d^{(s)}(i,k) - d^{(t)}(k,j+1) \\ & \le d^{(s)}(i+1,k+1) + d^{(t)}(k+1,j+1) - d^{(s)}(i+1,k) - d^{(t)}(k,j+1). \end{aligned} \tag{5}$$

That means we need to prove

$$d^{(s)}(i,k+1) - d^{(s)}(i,k) \le d^{(s)}(i+1,k+1) - d^{(s)}(i+1,k).$$

It holds by the iQL of $d^{(s)}(\cdot,\cdot)$ for $i \le i+1 \le k \le k+1$. □

(e) It is similar to the exponentiation by squaring method to calculate $d^{(r)}(\cdot,\cdot)$. Initially, we set $x = 1$ and calculate $d^{(x)} = d(\cdot,\cdot)$ in $O(n^2)$ time. Then, we write r in binary, and scan it from left to right from the second position.

- If the binary code we scan is 1, let $x' = 2x+1$, we calculate $d^{(2x)}$ from $d^{(x)}$ and then calculate $d^{(x')}$ from $d^{(2x)}$ and $d^{(1)}$, and then update $x = x'$.
- If the binary code we scan is 0, let $x' = 2x$, we calculate $d^{(2x)}$ by $d^{(x)}$, and then update $x = x'$.

Finally, we get $x = r$ and we get $d^{(r)}$ in $\log r$ rounds. Next, we give the DP algorithm to calculate $d^{(s+t)}$ by $d^{(s)}$ and $d^{(t)}$ in $O(n^2)$ time and so that our algorithm totally runs in $O(\log r \cdot n^2)$.

Algorithm 4: calculate $d^{(x)}$ from $d^{(s)}$ and $d^{(t)}$

```
1  ∀1 ≤ i ≤ n, d[x, i, i] ← 0 ;
2  ∀1 ≤ i ≤ n, K[i, i] ← i ;
3  for l = 2 to n − 1 do
4      for i = 1 to n − l + 1 do
5          j ← i + l − 1.;
6          d[x, i, j] ← −∞;
7          for k′ = K[i, j − 1] to K[i + 1, j] do
8              if d[s, i, k′] + d[t, k′, j] > d[x, i, j] then
9                  K[i, j] ← k′;
10                 d[x, i, j] ← d[s, i, k′] + d[t, k′, j]
11             end
12         end
13     end
14 end
```

Remark that we can conclude the algorithm above runs in $O(n^2)$ time because for each l, the "If" subroutine runs

$K[2][l] - K[1][l-1] + K[3][l+1] - K[2][l] ... + K[n-l+2][n] - K[n-l+1][n+1]$ times.

It equals to $K[n-l+2][n] - K[1][l-1] \leq n$, so the algorithm runs in $O(n^2)$.

Algorithm Design and Analysis
Assignment 6

1. You are given a set S of football teams, where each team $x \in S$ has already accumulated w_x wins so far. For every pair of teams $x, y \in S$ we know that there are g_{xy} games remaining between x and y (that is, if x wins a remaining games against y, then y must win $g_{xy} - a$ remaining games against x). Given a specific team $z \in S$, we would like to decide if z still has a chance to have the maximum number of wins. Alternatively, suppose we have the power to determine who wins each of the remaining games, is there a way such that z would get as many wins as anybody else by the end of the tournament? Give a polynomial-time algorithm for this problem. Hint: We can assume without loss of generality that z wins all remaining games. Is there a way to split the wins of the remaining games among the rest of the teams so that none of them get more wins than z? Consider a reduction to the network flow problem.

   Solution. Wlog, we can assume team z wins all the games left and only consider the other games. At first, let us calculate the number of wins z can make finally (use w_z^* to denote the number and $w_z^* = w_z + \sum_{(x \in S)} g_{xz}$). Then, we reduce the problem to a max flow problem by constructing the following network G. (Refer to Figure 1)

   1. Construct a source and a sink vertex s and t.
   2. For each team x except z, make a vertex v_x and connect it to t via an arc (v_x, t) with capacity $w_z^* - w_x$.
   3. For each team pair (x, y) $(x, y \neq z)$ we make a vertex M_{xy} and connect it to v_x and v_y via arcs (M_{xy}, v_x) and (M_{xy}, v_y) (infinity capacity). We also make an arc (s, M_{xy}) with capacity g_{xy} from the source.

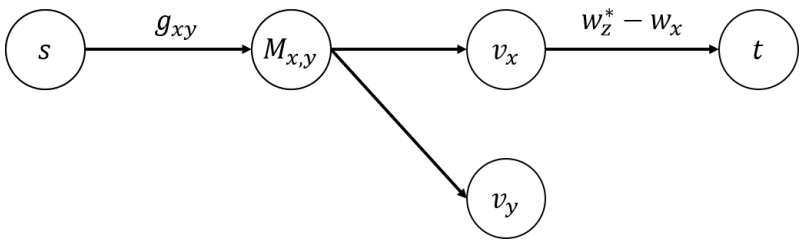


Figure 1: The example of the network.

Roughly speaking, a set of game results can be viewed as a balanced flow in the aforementioned network. One unit of flow goes from s to t via a path (s, M_{xy}, x, t) can be viewed as one game result that x wins one game between x and y. Therefore, consider a flow and a set of game results it represents, the flow on arcs (s, M_{xy}) can be viewed as

the number of game results between x and y and the flow on arcs (v_x, t) can be viewed as the number of wins of x. The capacity of (v_x, t) can control the number of wins of x, so that it can not exceed w_z^*, hence it controls the game results are feasible (i.e. i.e. no one wins more than z). Moreover, if the flow is full on each arc (s, M_{xy}) (i.e. $f(e = (s, M_{xy})) = g_{xy}$), it means that the flow can represent a set of game results that contains all the remaining games.

Therefore, we can run the max flow algorithm on G, if and only if all arcs (s, M_{xy}) are full in the max flow, we can find a feasible set of game results that contains all the remaining games. □

2. Let G be a flow network in which each edge e has a capacity $c(e)$ as well as a lower bound $d(e)$ on the flow it must carry. Note that $d(e) \leq c(e)$. A feasible flow assigns a value within the range of $[d(e), c(e)]$ to each edge e. Note that assigning a zero flow to every edge may not give a feasible flow of G (due to the lower bound requirement of the edges). Show how to make use a maximum flow algorithm to find a feasible flow of G.

   Solution. Let us construct a initialized flow f_0 such that each edge's flow equals to it's lower bound $d(e)$. ($f_0(e) = d(e)$) The problem is that it may be not a feasible flow because the flow of some vertices may be unbalanced. Consider an unbalanced vertex v, let $f_0^{in}(v)$ be the flow go into v and $f_0^{ou}(v)$ be the flow go out of v. Intuitively, to make the flow balanced, if $f_0^{ou}(v) > f_0^{in}(v)$, we should have additional $f_0^{ou}(v) - f_0^{in}(v)$ unit of flow go into vertex v in G. Symmetrically, if $f_0^{ou}(v) < f_0^{in}(v)$, we should have additional $f_0^{in}(v) - f_0^{ou}(v)$ unit of flow go out of vertex v in G.

   Follow this intuition, we create a network G' as follows (Refer to Figure 2). Roughly speaking, we want to use it to find a compensation that can make f_0 feasible.

   1. Make a source s and a sink t.
   2. Copy all vertices and edges from G to G', and set the capacity of each edge e to be $c(e) - d(e)$. (No longer lower bound)
   3. For each vertex v with $f_0^{in}(v) > f_0^{ou}(v)$, make a compensation arc (s, v) with capacity $f_0^{in}(v) - f_0^{ou}(v)$.
   4. For each vertex v with $f_0^{ou}(v) > f_0^{in}(v)$, make a compensation arc (v, t) with capacity $f_0^{ou}(v) - f_0^{in}(v)$.

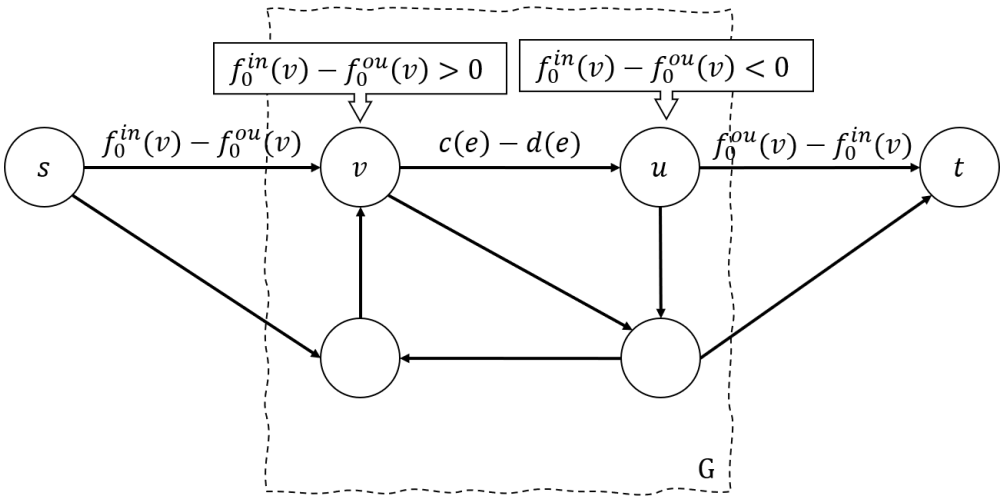


Figure 2: The example of the network.

We claim that G has a feasible flow if and only if all compensation arcs in G' are fully filled in the max flow of G'.

proof of the claim ($\rightarrow$): If all compensation arcs are full in the max flow f_1 of G', then for each edge e in G, we set $f_2(e) = f_1(e) + f_0(e)$. We have $d(e) \leq f_2(e) \leq c(e)$ simply because the $f_1(e)$ is non-negative and does not exceed $c(e) - d(e)$. Moreover, each vertex is balanced because

$$f_2^{in}(v) - f_2^{ou}(v) = f_1^{in}(v) - f_1^{ou}(v) + f_0^{in}(v) - f_0^{ou}(v) + \begin{cases} -f_1(e = (s, v)) & f_0^{in}(v) - f_0^{ou}(v) > 0 \\ +f_1(e = (v, t)) & f_0^{in}(v) - f_0^{ou}(v) < 0 \\ 0 & f_0^{in}(v) - f_0^{ou}(v) = 0. \end{cases}$$

Notice that it equals zero in all the cases, and hence f_2 is balanced on all vertices.

($\leftarrow$): On the other hand, if we have a feasible flow f_2 for G, we will construct the following flow f_1 in G'. (we will prove it's a feasible flow in G' where all compensation arcs are full, and hence it's a max flow.)

For each e in G, let $f_1(e) = f_2(e) - d(e) = f_2(e) - f_0(e)$. For each compensation arc (s, v) (or (v, t)), we set them be full (i.e. $f_1(e) = f_0^{in}(v) - f_0^{ou}(v)$ (or $f_0^{ou}(v) - f_0^{in}(v)$)). For each v other than s and t in G', if $f_0^{in}(v) - f_0^{ou}(v) > 0$, $f_1^{in}(v) - f_1^{ou}(v) = f_2^{in}(v) - f_2^{ou}(v) - f_0^{in}(v) + f_0^{ou}(v) + f_1(e = (s, v)) = 0$. We have it equals to 0 because $f_1(e = (s, v)) = f_0^{in}(v) - f_0^{ou}(v)$. For those v such that $f_0^{ou}(v) - f_0^{in}(v) < 0$, we can also prove $f_1^{in}(v) - f_1^{ou}(v) = 0$ symmetrically. Therefore, we prove that if there is a feasible flow in G, there is at least a feasible flow where all compensation arcs are full, such as f_1.

□

3. How long does it take you to finish the assignment (include thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty.

Algorithm Design and Analysis
Assignment 7

1. Show that the following problem is NP-Complete.

   MAXIMUM COMMON SUBGRAPH Input: Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$; a budget b. Output: Decide whether there exist two set of nodes $V_1' \subseteq V_1$ and $V_2' \subseteq V_2$ whose deletion leaves at least b nodes in each graph, and makes the two graphs identical.

   Solution. Given G_1, G_2, V_1', and V_2', we can efficiently delete these vertices from G_1 and G_2 and check the two remaining subgraph are identical and have at least b nodes in $O(n^2)$ time. (Notice that identical is different from isomorphic.) It proves that MAXIMUM COMMON SUBGRAPH is in NP.

   Consider the NP-Complete problem k-CLIQUE, which ask us to find a size k clique in a given graph G. Let us consider the MAXIMUM COMMON SUBGRAPH where $G_1 = G$, $b = k$, and G_2 be the size k clique. For any yes instance of the k-CLIQUE, we can find a size k clique, which is a size k subgraph of G_1 and is identical to G_2. Therefore, the input we construct before is also a yes instance of MAXIMUM COMMON SUBGRAPH. On the other hand, if the input of MAXIMUM COMMON SUBGRAPH is yes, then because the subgraph contains at least b nodes, it must be G_2, and G_2 must be a subgraph of G_1, which is a size k clique. Therefore, G should be a yes instance of k-CLIQUE. Thus far, we give a karp reduction from k-CLIQUE to MAXIMUM COMMON SUBGRAPH, and prove MAXIMUM COMMON SUBGRAPH is NP-Complete. □

2. Single execution time scheduling (SS). Given a set S of n jobs, a relation $\prec$ on S, a number of processors k and a time limit t. Does there exist a function f from S to $\{0, 1, ..., t-1\}$ such that

   1. $f^{-1}(i)$ has at most k members, and
   2. if $J \prec J'$, then $f(J) < f(J')$?

   For a verbal description of the problem, we need to assign those n jobs to those k processors. Each processor takes one unit of time to complete each job. $f(J)$ is the starting time for job J, and job J ends at time $f(J) + 1$. The first requirement above says that at most k jobs can be executed simultaneously. For two jobs J and J', $J \prec J'$ indicates J must be finished before executing J'. This is captured by the second requirement. We are deciding if we can schedule the n jobs on those k processors such that all of them are finished before time t.

   Now consider a more complex version: Single execution time scheduling with variable number of processors (SSV). Given a set S of n jobs, a relation $\prec$ on S, a time limit t, and a sequence of integers $c_0, c_1...., c_{t-1}$, where $\sum_{i=0}^{t-1} c_i = n$, does there exist a function f from S to $0, 1, ..., t-1$ such that

   1. $f^{-1}(i)$ has exactly c_i members, and
   2. if $J \prec J'$, then $f(J) < f(J')$?

   Show that (SS) is NP-Complete. You may follows the three steps below:

   (a) Show (SS) is in NP.

   (b) Show that there is a Karp reduction from SSV to SS: SSV $\leq_k$ SS.

   (c) Show that there is a Karp reduction from 3SAT to SSV: 3SAT $\leq_k$ SSV.

   Hint for part (c):

   - For each variable x_i $(i = 1, \ldots, n)$ in the 3SAT instance, create two gadgets corresponding to the two Boolean assignments to the variable as follows:
     - Gadget for $x_i =$ **true**: create $n + 2$ jobs $x_{i0}, x_{i1}, \ldots, x_{in}, y_i$ with $x_{i0} \prec x_{i1} \prec \cdots \prec x_{in}$ and $x_{i(i-1)} \prec y_i$;
     - Gadget for $x_i =$ **false**: create $n + 2$ jobs $\bar{x}_{i0}, \bar{x}_{i1}, \ldots, \bar{x}_{in}, \bar{y}_i$ with $\bar{x}_{i0} \prec \bar{x}_{i1} \prec \cdots \prec \bar{x}_{in}$ and $\bar{x}_{i(i-1)} \prec \bar{y}_i$.
     - Intuitive idea: we may imagine x_i (or $\bar{x}_i$) to be true if and only if x_{i0} (or $\bar{x}_{i0}$, respectively) is executed at time 0, all the other jobs are used to control the requirement of the 3SAT problem.

- For each clause C_j ($j = 1, \ldots, m$), create 7 jobs $c^j_{ttt}, c^j_{ttf}, c^j_{tft}, c^j_{ftt}, c^j_{tff}, c^j_{ftf}, c^j_{fft}$ corresponding to the seven possibilities of the values for the three literals in the clause (that makes the clause evaluated to **true**). For example, c^j_{ttf} corresponds to that the first and the second literals are **true** and the third literal is **false**. Then, link each of the seven jobs to the last job in the variable gadget accordingly. For example, if we have a clause $C_j = x_3 \vee \bar{x}_4 \vee x_7$ and consider the job c^j_{ttf} (which corresponds to $x_3 =$ **true**, $\bar{x}_4 =$ **true** and $x_7 =$ **false**), we have $x_{3n} \prec c^j_{ttf}$, $\bar{x}_{4n} \prec c^j_{ttf}$ and $\bar{x}_{7n} \prec c^j_{ttf}$.
- Consider $t = n + 2$. Setup the values for $c_0, c_1, \ldots, c_{t-1}$ appropriately such that, in order to finish all the tasks, we must have the followings:
  - For each $i = 1, \ldots, n$, we must have either $f(x_{i0}) = 0, f(\bar{x}_{i0}) = 1$ or $f(x_{i0}) = 1, f(\bar{x}_{i0}) = 0$. The former case will represent $x_i =$ **true** and the latter case will represent $x_i =$ **false**;
  - For the 7 jobs corresponding to each clause, exactly one job must be executed at time n, and the remaining 6 jobs must be executed at time $n + 1$. The job executed at time n will represent the values for the three literals in the clause in a satisfying Boolean assignment.

Solution. Please refer to the paper of J.D. ULLMAN. (P2) is our problem (SS) in the paper, they prove (P2) is NP-Complete via (P4) (i.e. (SSV)). □

3. How long does it take you to finish the assignment (include thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty.