Bo Yang (Aaron) USCID: 7526922531

USC CSCI 570 Homework1 Date: 2020-07-03

1. Graded Problems

1.State True/False: An instance of the stable marriage problem has a unique stable matching if and only if the version of the Gale-Shapely algorithm where the male proposes and the version where the female proposes both yield the exact same matching.

> ***True***
> Given the Gale-Shapely algorithm, every man will propose a woman based on his preference list. If the woman ***w*** is the first woman in man ***m's*** preference list. man ***m*** will propose to woman ***w*** without hesitation. Luckily, man ***m*** is also the first choice of woman ***w***, since man ***m*** ranked first in her preference list. So they will have a deal. When other men propose to woman ***w***, ***w*** will reject. Since there is no one's ranking is higher than man ***m*** in her preference list. So we can come to the conclusion that this statement is true.

2. A stable roommate problem with 4 students a, b, c, d is defined as follows. Each student ranks the other three in strict order of preference. A matching is defined as the separation of the students into two disjoint pairs. A matching is stable if no two separated students prefer each other to their current roommates. Does a stable matching always exist? If yes, give a proof. Otherwise give an example roommate preference where no stable matching exists.

> ***A stable matching might not exist***
> Here is a counterexample.

```
[
    a: [b, c, d],
    b: [c, a, d],
    c: [a, b, d],
    d: [a, b, c]
]
```

> Based on this preference list, we can run the Gale-Shapely algorithm to find the *' stable matching '*. And finally, we get a result pairs => [(***b***, ***c***), (***a***, ***d***)]. But pair (***a***, ***d***) is not stable in this context since for anyone who dated with ***d***, another member has a higher rank over ***d***, implying that ***d's*** current roommate prefers other than ***d***, and this relationship will break. ***E.G.*** (***a***, ***c***) is a more favorable pairing than (***a***, ***d***). If we keep pair (***a***, ***c***), the (***b***, ***d***) will has the same problem. they won't stop. So, we have to say, there is no stable matching in this given context.

3. Solve Kleinberg and Tardos, Chapter 1, Exercise 4.

Gale and Shapley published their paper on the Stable Matching Problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.

Basically, the situation was the following. There were ***m*** hospitals, each with a certain number of available positions for hiring residents. There were ***n*** medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference.

We will assume that there were more students graduating than there were slots available in the ***m*** hospitals.

The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.)

We say that a set of assignments of students to hospitals is *stable* if neither of the following situations arises

We say that an assignment of students to hospitals is stable if neither of the following situations arises.

- First type of instability: There are students ***s*** and ***s′***, and a hospital ***h***, so that
  - ***s*** is assigned to ***h***, and
  - ***s′*** is assigned to no hospital, and
  - ***h*** prefers ***s′*** to ***s***.
- Second type of instability: There are students ***s*** and ***s′***, and hospitals ***h*** and ***h′***, so that
  - ***s*** is assigned to ***h***, and
  - ***s′*** is assigned to ***h′***, and
  - ***h*** prefers ***s′*** to ***s***, and
  - ***s′*** prefers ***h*** to ***h′***.

So we basically have the Stable Matching Problem, except that

- (i) hospitals generally want more than one resident, and
- (ii) there is a surplus of medical students.

Show that there is always a stable assignment of students to hospitals, and give an algorithm to find one.

> ***Answer:***
> This hospital positions and students matching problem is similar to the marriage matching problem. So base on the same logic, we can check whether the remaining position slots are zero in each iteration. and accept or reject every student in its preference list.
>
> so the pseudocode will be this:

```
while hospital h still has some available positions:
    invite top preference student s join hospital h

    if student sis avaiable:
        s accept the offer, and go to h
    else:
        # decision power is on student's hands
        # denote 'new_h' is the new hospital who trying to hire he/her
        # denote 'cur_h' is the current hosptial which student s is working for right now
        compare both hosptial(new_h, cur_h) ranking in student s preference list
        if new_h`s ranking is higher than cur_h`s ranking:
            quit and accpet new offer
        else:
            reject and still working for current hospital
        end if
    end if
end while
```

4. N men and N women were participating in a stable matching process in a small town named Walnut Grove. A stable matching was found after the matching process finished and everyone got engaged. However, a man named Almazo Wilder, who is engaged with a woman named Nelly Oleson, suddenly changes his mind by preferring another woman named Laura Ingles, who was originally ranked right below Nelly in his preference list, therefore Laura and Nelly swapped their positions in Almanzo's preference list. Your job now is to find a new matching for all of these people and to take into account the new preference of Almanzo, but you don't want to run the whole process from the beginning again, and want to take advantage of the results you currently have from the previous matching.

Describe your algorithm for this problem. Assume that no woman gets offended if she got refused and then gets proposed by the same person again.

> ***Answer:***
> There are at least two suitation might happen:
>
> 1. Almanzo will accpeted by Laura.

> when Almanzo broke up with Nelly (his current girlfriend) and proposed to Laura, and Laura accepted it. This would cause Laura has to break up with his current boyfriend. This will definitely make an instabilities problem. like a chain reaction, other's relationship breakdown will affect the rest people's marriages. So if we don't want to build the algorithm from scratch, we need to collect these unlucky men into the 'free pool' again and run the G-S algorithm until every man engages with somewhat new a woman. This could might lead to bigger chaos, and might need to rearrange all people's matching in the worst case. Luckily, this matching process still has a possibility that will end very soon. if Nelly is willing to accept Laura's ex-boyfriend.

2. Almanzo will not accpeted by Laura.

> when Almanzo were rejected by Laura, there is no candidate for him to choose except his ex-girlfriend, Nelly. So he has to go back to his home and beg Nelly's forgiveness. We don't need to run the algorithm. and this will be a very sad story.

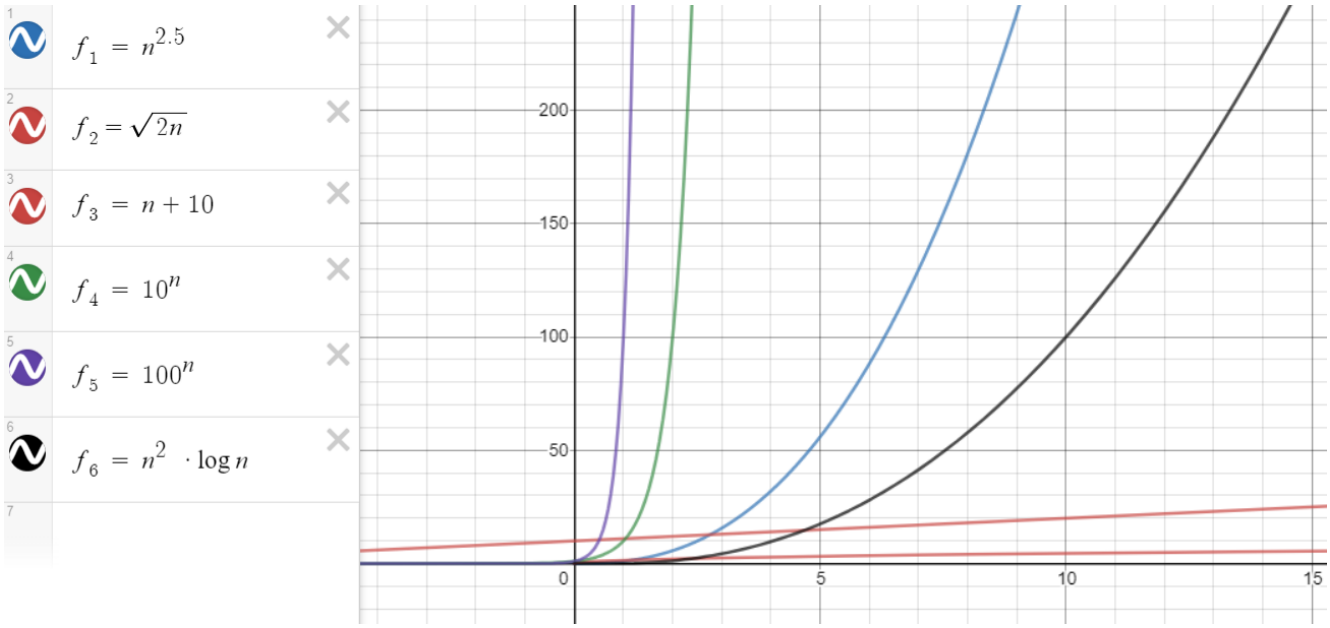
5. Solve Kleinberg and Tardos, Chapter 2, Exercise 3.

Take the following list of functions and arrange them in ascending order of growth rate. That is, if function ***g(n)*** immediately follows function ***f(n)*** in your list, then it should be the case that ***f(n)*** is ***O(g(n))***

$f_1(n) = n^{2.5}$
$f_2(n) = \sqrt{2n}$
$f_3(n) = n + 10$
$f_4(n) = 10^n$
$f_5(n) = 100^n$
$f_6(n) = n^2 * logn$

> in ascending order of growth, the result list will be $f_2, f_3, f_6, f_1, f_4, f_5$

and



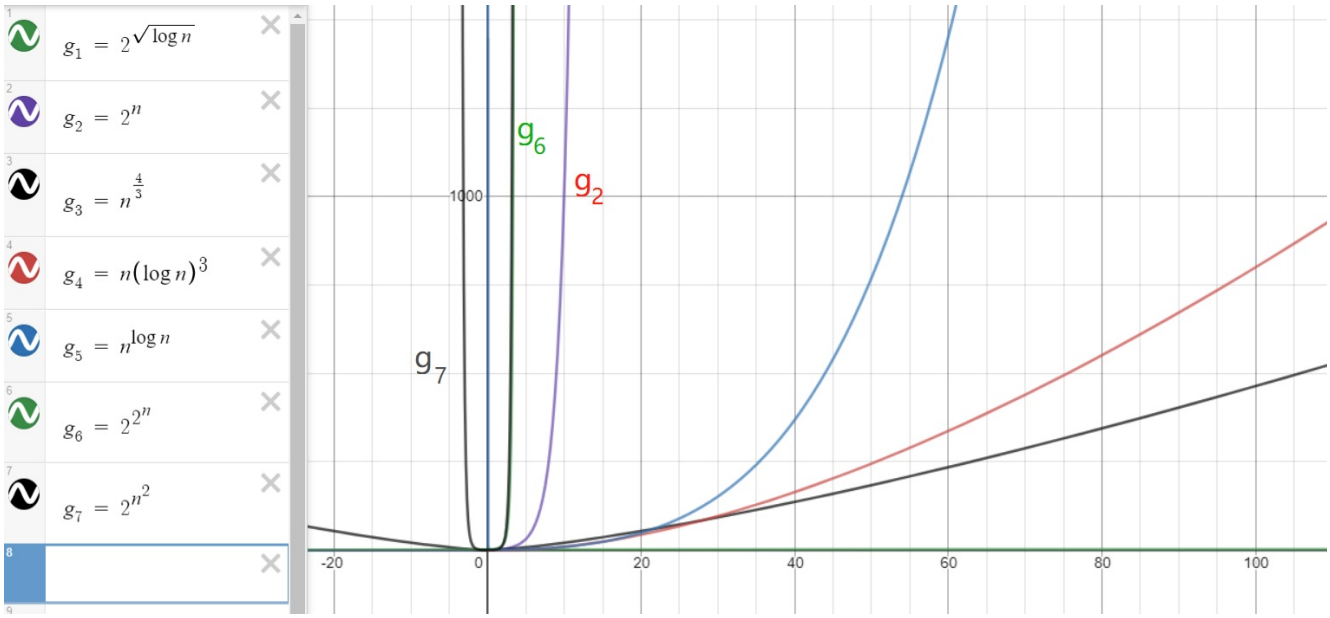
6. Solve Kleinberg and Tardos, Chapter 2, Exercise 4.

Take the following list of functions and arrange them in ascending order of growth rate. That is, if function ***g(n)*** immediately follows function ***f(n)*** in your list, then it should be the case that ***f(n)*** is ***O(g(n))***.

$g_1(n) = 2^{\sqrt{logn}}$
$g_2(n) = 2^n$

$g_3(n) = n^{4/3}$
$g_4(n) = n(logn)^3$
$g_5(n) = n^{logn}$
$g_6(n) = 2^{2^n}$
$g_7(n) = 2^{n^2}$

> in ascending order of growth, the result list will be $g_1 g_3, g_4, g_5, g_2, g_7, g_6$



7. Solve Kleinberg and Tardos, Chapter 2, Exercise 5.

Assume you have functions ***f*** and ***g*** such that ***f(n)*** is ***O(g(n))***. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

1. $log_2 f(n)$ is $O(log_2 g(n))$.
2. $2^{f(n)}$ is $O(2^{g(n)})$.
3. $f(n)^2$ is $O(g(n)^2)$.

1. ***False***

> As we know, the big O annotation is the asymptotic upper bound. $f(n) = O(g(n))$. if we have $f(n) = c$, a constant number, which is greater than 2. $g(n) = 1$, since $c = O(1)$.
> $log_2 c$ is not equal to $log_2 1 = 0$. Therefore, we cannot say $log_2 f(n) = O(log_2 g(n))$.
> But if there exists some ***n***, make the $g(n)$ greater than 2, $O(log_2 g(n))$ won't equal to ***zero 0*** any more, at that time, we can say, this statement is True.

1. ***False***

> For the same reason, $f(n) = O(g(n))$. let's say $f(n) = 2n$, so the $g(n) = n$. since $2n = O(n)$, right. next, we take these number back to our statement. $2^{f(n)} = 2^{2n} = 4^n$ and $2^{g(n)} = 2^n$.
> we cannot say $4^n = O(2^n)$, since $4^n = O(4^n)$

2. ***True***

> For the same reason, $f(n) = O(g(n))$. there exists positive constants ***c*** and n_0 such that $f(n) \leq cg(n);\ \forall n \geq n_0$. This implies $f(n)^2 \leq c^2 g(n)^2;\ \forall n \geq n_0$, which in turn implies that $f(n)^2 = O(g(n)^2)$.

8. Which of the following statements are true?

1. If f, g, and h are positive increasing functions with f in **O(h)** and g in **Ω(h)**, then the function $f + g$ must be in **Θ(h)**.
2. Given a problem with input of size ***n***, a solution with **O(n)** time complexity always costs less in computing time than a solution

with **O(n^2)** time complexity.
3. $F(n) = 4n + \sqrt{3n}$ is both **O(n)** and **Θ(n)**.
4. For a search starting at node s in graph G, the DFS Tree is never as the same as the BFS tree.
5. BFS can be used to find the shortest path between any two nodes in a non-weighted graph

1. ***False***

> The ***Theorem 2.4*** said that ***f*** and ***g*** are two functions such that for some other function ***h***, we have ***f = O(h)*** and ***g = O(h)***. Then ***f + g = O(h)***

2. ***False***

> The ***O(n)*** annotation just tells you the maxinum of time that your algorithm might cost. if we really want to compare which algorithmes runs fast. we also need to provide the lower bound annotation **Ω**.

3. ***True***

> Since The dominant term is $4n$, which is obviously both $O(n)$ and $\Omega(n)$.

4. ***False***

> if the graph ***G*** only has three nodes ∈ {***A, B, C***}. and its edges are ***A -> B*** and ***A -> C***.
> using BFS or DFS to traverse G will generate the same result.

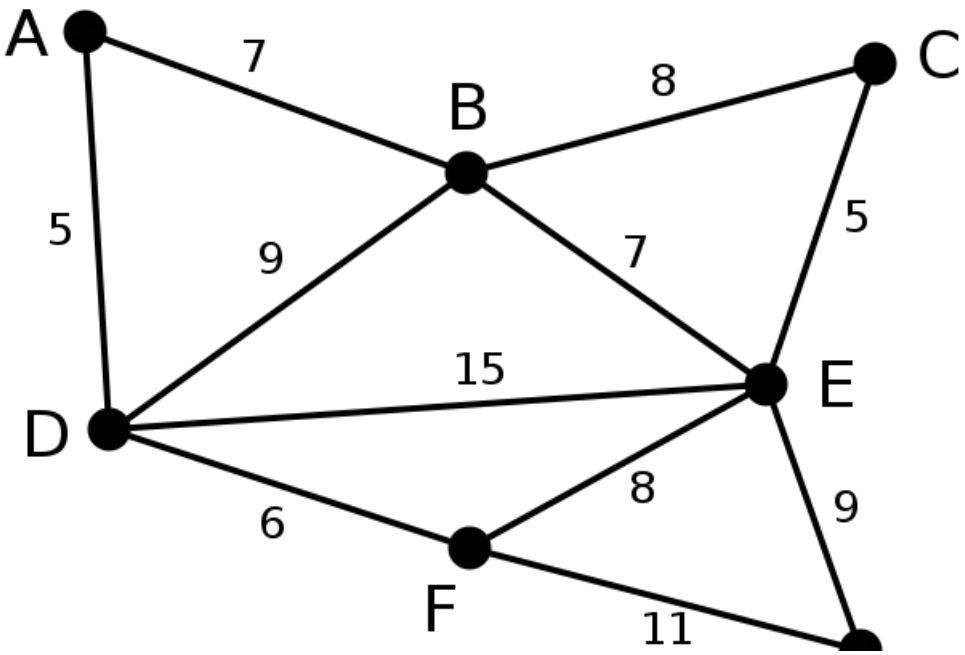
5. ***True***

> That's the one problem that BFS trying to solve.

9. Solve Kleinberg and Tardos, Chapter 3, Exercise 2.

Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. (It should not output all cycles in the graph, just one of them.) The running time of your algorithm should be ***O(m + n)*** for a graph with ***n*** nodes and ***m*** edges.

> ***Answer:***
> Based on this article said, we can use disjoint-set to store graph, and use union-find algoirthm to detect whether an undirected graph contains a cycle or not.



> the following python code has been tested.

```python
def cmp(key1, key2):
    return (key1, key2) if key1 < key2 else (key2, key1)


def find_parent(record, node):
    if record[node] != node:
        record[node] = find_parent(record, record[node])
    return record[node]


def naive_union(record, edge):
    u, v = find_parent(record, edge[0]), find_parent(record, edge[1])
    record[u] = v


def has_cycle(graph, init_node):
    edge_dict = {}
    for node in graph.keys():
        edge_dict.update({cmp(node, k): v for k, v in graph[node].items()})
    connected_records = {key: key for key in graph.keys()}

    for edge_pair in list(edge_dict.keys()):
        if find_parent(connected_records, edge_pair[0]) != \
                find_parent(connected_records, edge_pair[1]):
            naive_union(connected_records, edge_pair)
        else:
            return True
    return False

if __name__ == '__main__':
    graph_dict = {
        "A": {"B": 7, "D": 5},
        "B": {"A": 7, "C": 8, "D": 9, "E": 5},
        "C": {"B": 8, "E": 5},
        "D": {"A": 5, "B": 9, "E": 15, "F": 6},
        "E": {"B": 7, "C": 5, "D": 15, "F": 8, "G": 9},
        "F": {"D": 6, "E": 8, "G": 11},
        "G": {"E": 9, "F": 11}
    }

    print(has_cycle(graph_dict, "D"))  # True
```

2. Practice Problems

1. Reading Assignment: Kleinberg and Tardos, Chapters 1, 2, and 3.

2. Solve Kleinberg and Tardos, Chapter 1, Exercise 1.

Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

True or false? In every instance of the Stable Matching Problem, there is a stable matching containing a pair ***(m, w)*** such that ***m*** is ranked first on the preference list of ***w*** and ***w*** is ranked first on the preference list of ***m***.

False
E.G. man and woman has such preference list.
man $m \in \{m_0, m_1, m_2\}$ woman $w \in \{w_0, w_1, w_2\}$. (man propose)

$|w_0, w_1, w_2|$
$|w_1, w_2, w_0|$ => **man_preference**
$|w_2, w_0, w_1|$

$|m_2, m_1, m_0|$
$|m_0, m_2, m_1|$ => **woman_preference**
$|m_1, m_0, m_2|$

based on these two preference list, we can generate stable matching pair.
(m_0, w_0), (m_1, w_1), (m_1, w_1).
as you can see, the woman dated with each man is his first preference list, indeed. but those women are not very font of her current boyfriend. all of them put their boyfriend in the last position.

3. Solve Kleinberg and Tardos, Chapter 1, Exercise 2.

Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

True or false? Consider an instance of the Stable Matching Problem in which there exists a man ***m*** and a woman ***w*** such that ***m*** is ranked first on the preference list of ***w*** and ***w*** is ranked first on the preference list of ***m***. Then in every stable matching ***S*** for this instance, the pair ***(m, w)*** belongs to ***S***.

True

Suppose there were a pair ***(m, w)*** meet those prerequirements,***m*** is ranked first on the preference list of ***w*** and ***w*** is ranked first on the preference list of ***m***, but in the end, the stable matching result didn't contain ***(m, w)***.

when ***m*** turns to propose, ***w*** will be his first choice, since woman ***w*** ranked first in his list. ***(m, w)*** generated. if no other men propose to her, the ***(m, w)*** should appear in the result. if another wooer appeared, the women ***m*** will compare both admirers, since the current boyfriend ***m*** ranked first in her list. so she will reject other's propose. the pair ***(m, w)*** will be survived. both situations will contract our assumption.

4. Solve Kleinberg and Tardos, Chapter 1, Exercise 3.

There are many other settings in which we can ask questions related to some type of "stability" principle. Here's one, involving competition between two enterprises.

Suppose we have two television networks, whom we'll call ***A*** and ***B***. There are ***n*** prime-time programming slots, and each network has ***n*** TV shows. Each network wants to devise a schedule—an assignment of each show to a distinct slot—so as to attract as much market share as possible.

Here is the way we determine how well the two networks perform relative to each other, given their schedules. Each show has a fixed rating, which is based on the number of people who watched it last year; we'll assume that no two shows have exactly the same rating. A network wins a given time slot if the show that it schedules for the time slot has a larger rating than the show the other network schedules for that time slot. The goal of each network is to win as many time slots as possible.

Suppose in the opening week of the fall season, Network ***A*** reveals a schedule ***S*** and Network ***B*** reveals a schedule ***T***. On the basis of this pair of schedules, each network wins certain time slots, according to the rule above. We'll say that the pair of schedules ***(S, T)*** is stable if neither network can unilaterally change its own schedule and win more time slots. That is, there is no schedule ***S'*** such that Network ***A*** wins more slots with the pair ***(S', T)*** than it did with the pair ***(S, T)***; and symmetrically, there is no schedule ***T'*** such that Network ***B*** wins more slots with the pair ***(S, T')*** than it did with the pair ***(S, T)***.

The analogue of Gale and Shapley's question for this kind of stability is the following: For every set of TV shows and ratings, is there always a stable pair of schedules? Resolve this question by doing one of the following two things:

1. give an algorithm that, for any set of TV shows and associated ratings, produces a stable pair of schedules; or
2. give an example of a set of TV shows and associated ratings for which there is no stable pair of schedules.

> ***A stable matching might not exist***
> Here is a counterexample. assume there are only two slot avaiable. And network A and B has two shows in their schedule, respectively.
>
> ```
> [     slot1, slot2
>     A: [10,   20],
>     B: [15,   25]
> ]    B wins  B wins
> ```
>
> Based on this preference list, we can say that network B wins all two slots. But the network A change change its schedule unilaterally. its new schedule will be like this:
>
> ```
> [     slot1, slot2
>     A: [20,   10],
>     B: [15,   25]
> ]    A wins  B wins
> ```
>
> Now, network A and B has a tie, which also means that previous pair is not stable.

5. Solve Kleinberg and Tardos, Chapter 2, Exercise 6.

Consider the following basic problem. You're given an array A consisting of n integers ***A[1], A[2], ... A[n]***. You'd like to output a two-dimensional n-by-n array B in which ***B[i, j] (for i < j)*** contains the sum of array entries ***A[i]*** through ***A[j]***—that is, the sum ***A[i]+ A[i + 1]+ ... + A[j]***. The value of array entry ***B[i, j]*** is left unspecified whenever ***i ≥ j***, so it doesn't matter what is output for these values.) Here's a simple algorithm to solve this problem

```
For i = 1, 2, . . . , n
    For j = i + 1, i + 2, . . . , n
        Add up array entries A[i] through A[j]
        Store the result in B[i, j]
    Endfor
Endfor
```

1. For some function ***f*** that you should choose, give a bound of the form ***O(f(n))*** on the running time of this algorithm on an input of size ***n*** (i.e., a bound on the number of operations performed by the algorithm).
2. For this same function ***f***, show that the running time of the algorithm on an input of size ***n*** is also ***Ω(f(n))***. (This shows an asymptotically tight bound of ***Θ(f(n))*** on the running time.)
3. Although the algorithm you analyzed in parts (1) and (2) is the most natural way to solve the problem—after all, it just iterates through the relevant entries of the array ***B***, filling in a value for each—it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time ***O(g(n))***, where

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

1. This algoirthm will cost $O(n^3)$ at most.

> after we convert the pseudocode into python.
>
> ```python
> """
> arr is the variable which represent array A
> result is the dict which use to store the result, equals to B
> """
> for i in range(1, n + 1):
>     for j in range(i + 1, n + 1):
>         temp_sum = sum(arr[i:j])
>         result[(i,j)] = temp_sum
> ```
>
> it is easy to notice that there are two loops before doing the actual sum operation.
> when ***i = 1***, inner loop runs ***n-1*** times.

> when ***i = 2***, inner loop runs ***n-2*** times.
> ...
> when ***i = n - 2***, inner loop runs ***2*** time.
> when ***i = n - 1***, inner loop runs ***1*** time.
> when ***i = n***, inner loop runs ***0*** time.
> So, two loops will cost ***sum(n-1, n-2, ... 1, 0) =*** $0.5n^2 - 0.5n$ = ***O(n^2)*** time. and sum up the $a[i:j]$ will cost ***O(j-i+1)*** time. we just assume it will cost $O(n)$ time. About the assign the value ***temp_sum*** to the result, it costs only $O(1)$ time. Finally, this algoirthm will cost $O(n^3)$ at most.

2. This algoirthm's asymptotic lower bound is also $\Theta(n^3)$.

> I think there is no difference between $O(f(n))$ and $\Theta(f(n))$. so I think the asymptotic lower bound is $\Theta(n^3)$ as well.

3. Basic thought is that we can take advantage of previous sum result to reduce the time complexity. My new algorithm will only cost $O(n^2)$ time.

> we firstly compute the result of diagonal, and then use this result to fill out the upper triangular matrix.
>
> ```
> for i in range(1, n + 1):
>     result[(i, i+1)] = arr[i] + arr[i + 1]
> for k in range(2, n):
>     for i in range(1, n - k + 1):
>         j = i + k
>         result[(i, j)] = result[(i, j - 1)] + arr[j]
> ```
>
> in the first loop, iteration runs ***n*** times. and indexing in an array cost $O(1)$ time. so overall, first loop costs $O(n)$ time.
> in the second loop, the actual logic operation also get value by index, which spend $O(1)$ time, so we only need to focus how many loops do I have.
> when ***k = 2***, inner loop runs ***n-2*** times.
> when ***k = 3***, inner loop runs ***n-3*** times.
> ...
> when ***k = n - 2***, inner loop runs ***1*** time.
> when ***k = n - 1***, inner loop runs ***0*** time.
> So, the second loop will cost ***sum(n-2, n-3, ... 1, 0) =*** $0.5n^2 - 2n + 2$ = ***O(n^2)*** time. since the second loop is the dominant part, Finally, this algoirthm will cost $O(n^2)$ at most.

6. Solve Kleinberg and Tardos, Chapter 3, Exercise 6.

We have a connected graph ***G = (V, E)***, and a specific vertex ***u ∈ V***. Suppose we compute a depth-first search tree rooted at ***u***, and obtain a tree **T** that includes all nodes of **G**. Suppose we then compute a breadth-first search tree rooted at ***u***, and obtain the same tree **T**. Prove that ***G = T***. (In other words, if **T** is both a depth-first search tree and a breadth-first search tree rooted at **u**, then **G** cannot contain any edges that do not belong to **T**.)

> Let assume that ***G*** contains an edge ***e = (x,y)*** that not exist in result ***T***. Based on ***Theorem 3.7***, we know that one of ***x*** or ***y*** is an ancestor of the other, which means ***x*** and ***y*** are on different levels. Also, ***T*** is a BFS tree as well. given the ***Theorem 3.4***'s explanation, if edge ***(x,y)*** in the BFS tree, their level differ at most 1. therefore we can know their level differ by exactly 1. However, combining that one of ***x*** or ***y*** is the ancestor of the other and that their level differ by 1 implies that the edge ***(x, y)*** is in the tree ***T***, which contradicts our assumption.
>
> > ***Theorem 3.4*** Let T be a breadth-first search tree, let ***x*** and ***y*** be nodes in T belonging to layers L_i and L_j respectively, and let ***(x, y)*** be an edge of ***G***. Then ***i*** and ***j*** differ by at most 1.
> >
> > ***Theorem 3.7*** Let T be a depth-first search tree, let ***x*** and ***y*** be nodes in T, and let ***(x, y)*** be an edge of ***G*** that is not an edge of ***T***. Then one of ***x*** or ***y*** is an ancestor of the other