# Exam 1 Review Session

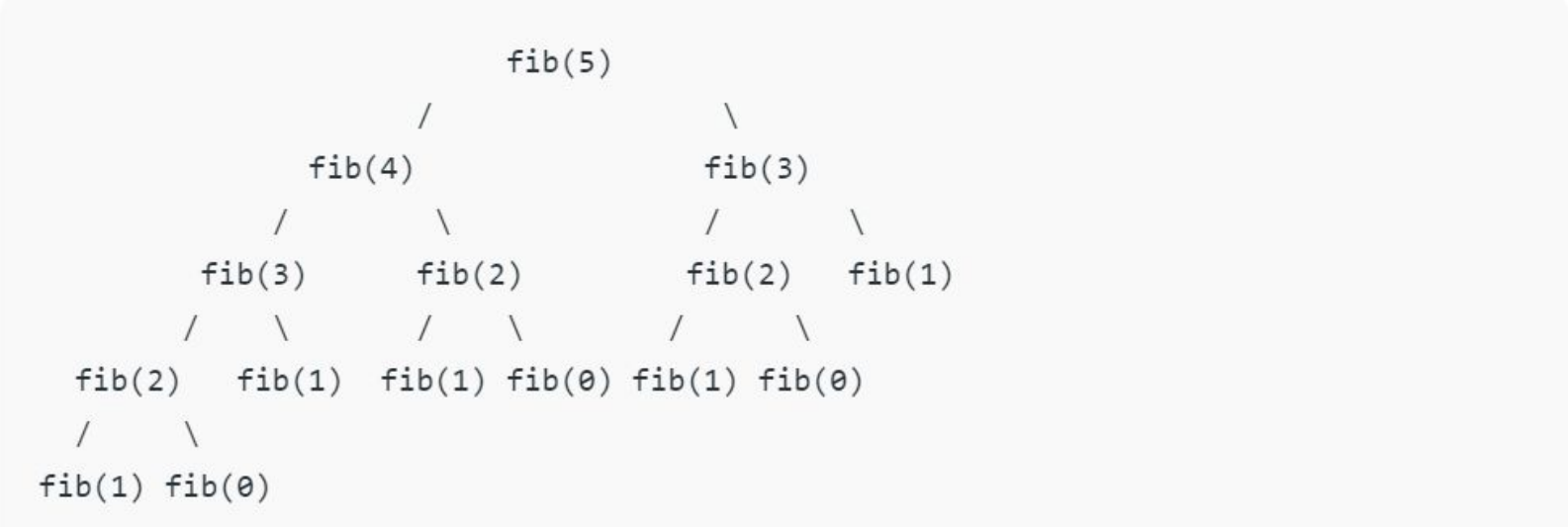
# Greedy Algorithm

Chi Zhang

# TRUE/FALSE

In a dynamic programming formulation, the sub-problems must be mutually independent.

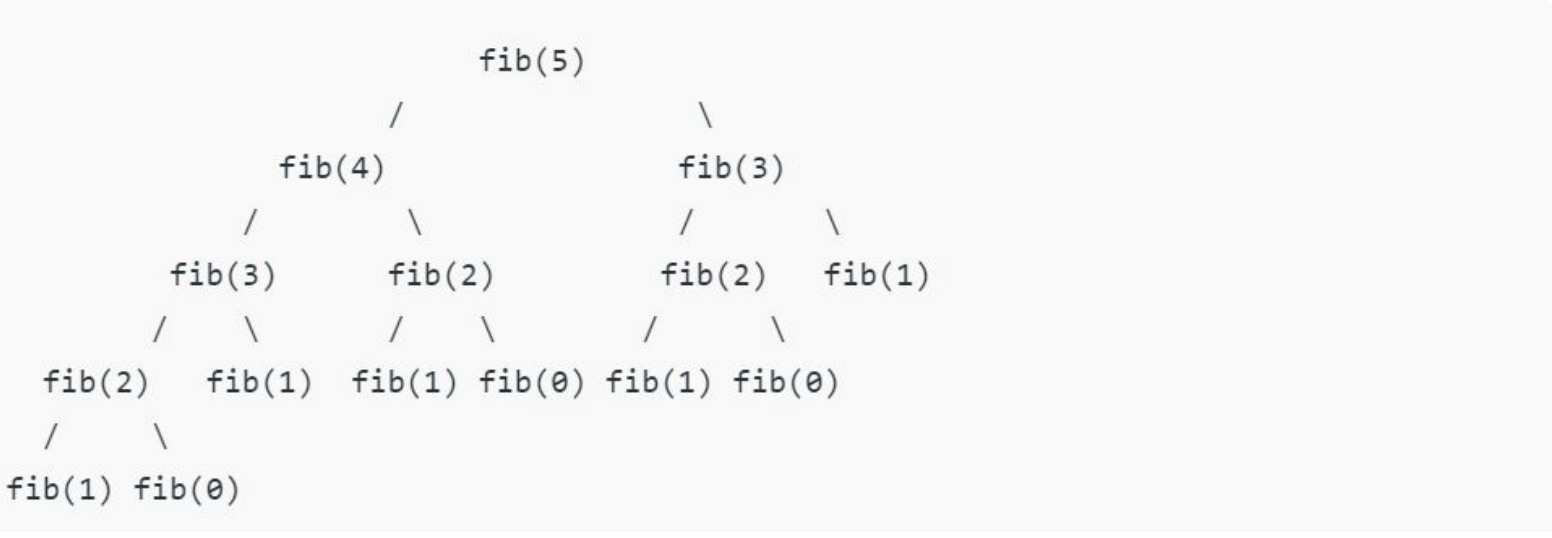
False! That is a requirement for divide and conquer, not dynamic programming.



# TRUE/FALSE

In dynamic programming, you must calculate the optimal value of a sub-problem twice, once during the bottom up pass and once during the top down pass.

False! Dynamic programming either uses the top-down approach (memoization) or the bottom-up one (tabulation), but not both.



# TRUE/FALSE

In a divide & conquer algorithm, the size of each sub-problem must be at most half the size of the original problem.

False
This is not a requirement for divide and conquer. However, divide and conquer requires them to be **mutually independent**.
For example, $T(n) = T(n/10) + T(n/10 * 9) + O(n)$
We will see an example in three slides.

# TRUE/FALSE

Amortized cost of operations in a Fibonacci heap is at least as good as the worst case cost of those same operations in a binomial heap.

True

| Procedure | Binary heap (worst-case) | Binomial heap (worst-case) | Fibonacci heap (amortized) |
|---|---|---|---|
| Make-Heap | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| Insert | $\Theta(\lg n)$ | $O(\lg n)$ | $\Theta(1)$ |
| Minimum | $\Theta(1)$ | $O(\lg n)$ | $\Theta(1)$ |
| Extract-Min | $\Theta(\lg n)$ | $\Theta(\lg n)$ | $O(\lg n)$ |
| Union | $\Theta(n)$ | $O(\lg n)$ | $\Theta(1)$ |
| Decrease-Key | $\Theta(\lg n)$ | $\Theta(\lg n)$ | $\Theta(1)$ |
| Delete | $\Theta(\lg n)$ | $\Theta(\lg n)$ | $O(\lg n)$ |

# TRUE/FALSE

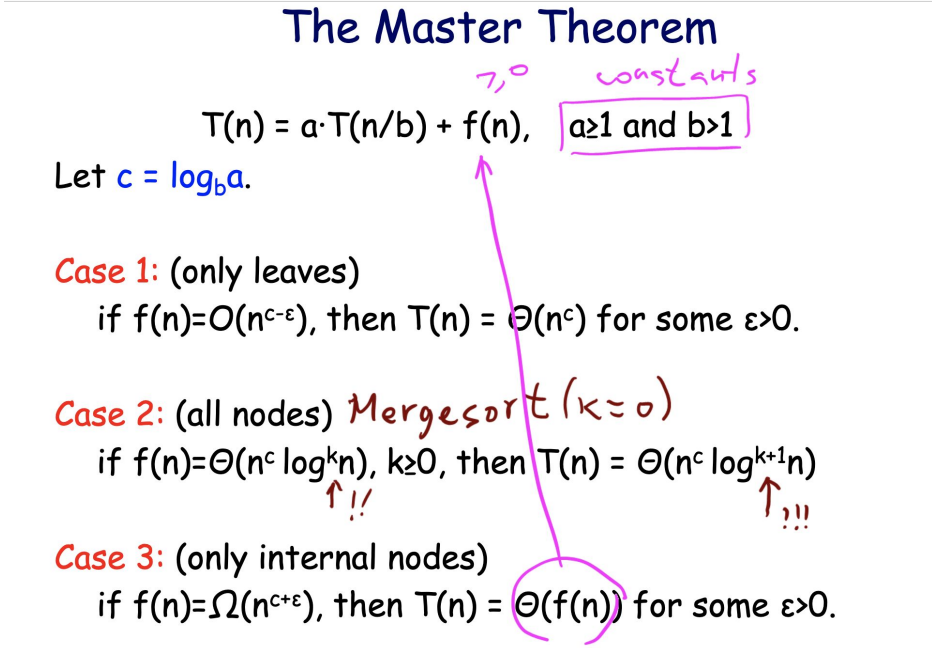
The Master theorem can always be used to find the complexity of T(n), if it can be written as a recurrence relation T(n) = aT($nb$ ) + f(n).

False!

$O(n^{c-\epsilon})$ $\Theta(n^c \log^k n)$ $\Omega(n^{c+\epsilon})$

Does case 1, 2, 3 cover every possibilities?

No! Similar to Case 2, but k < 0.
Other examples: $\Theta(n^c \log \log^k n)$



# TRUE/FALSE

Suppose we are given an array A of n distinct elements, and we want to find n/2 elements in the array whose median is also the median of A. Any algorithm that does this must take Ω(n log n) time.

Alternative solution without sorting:
Step 1: Find the median number: O(n) using SELECT
Step 2: Iterate over the list, select n/4 elements larger than the median and another n/4 elements smaller than the median. O(n)

SELECT: https://en.wikipedia.org/wiki/Median_of_medians

# QuickSort (Partition Sort)

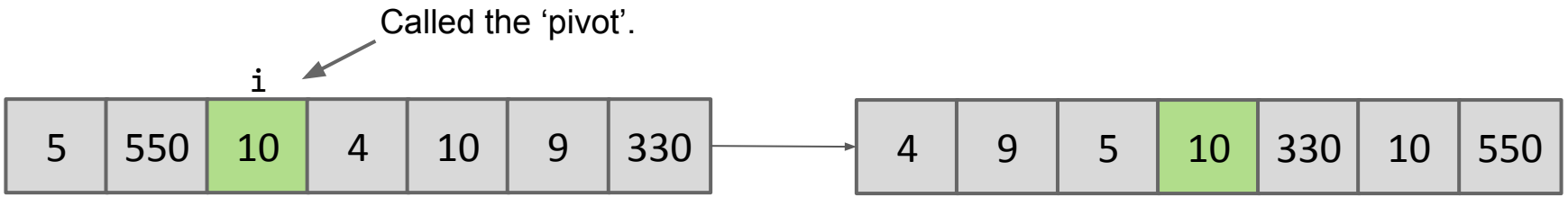
Core idea: To partition an array a[] on element x=a[i] is to rearrange a[] so that:

x moves to position j (may be the same as i)

- All entries to the left of x are <= x.
- All entries to the right of x are >= x.

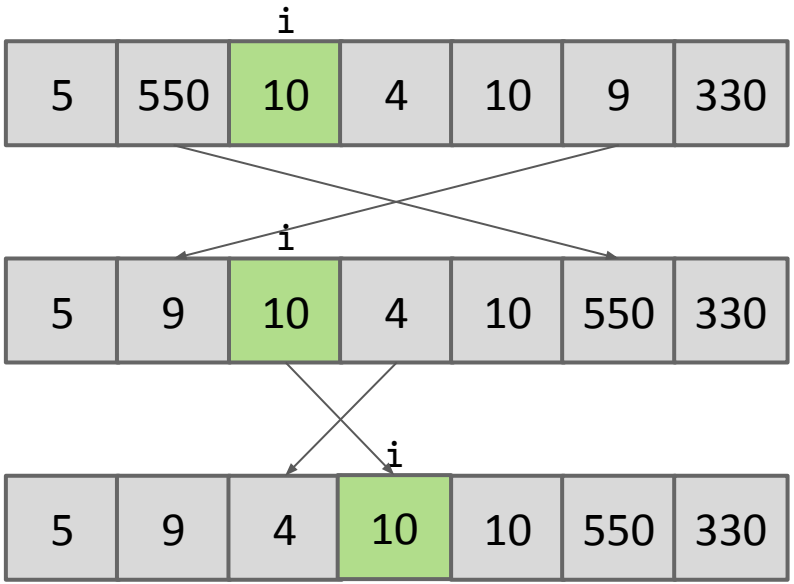
Observations:

- Element x is already in place.
- Recursively partition left and right



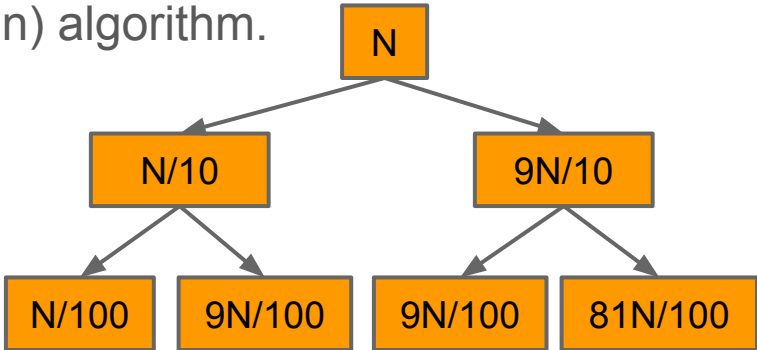
# Partition

- Scan the list once, swap the element larger than x with the last element smaller than x. Time complexity: O(N)



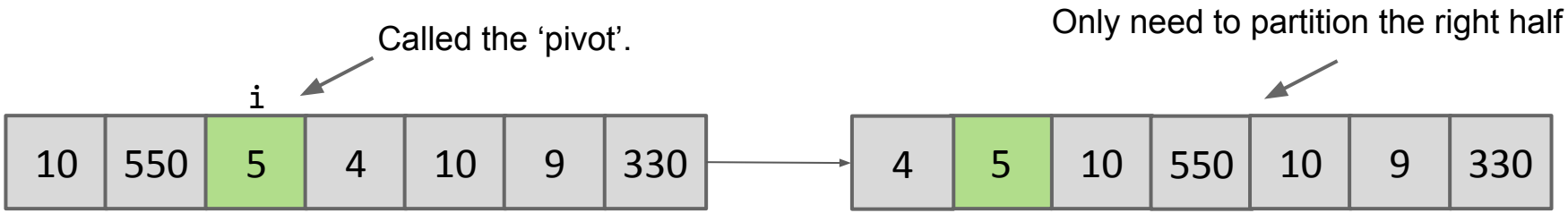
# QuickSort Time Complexity

- Time complexity of partition: O(N)
- Assume the pivot points partition the array into N-b and b.
- T(n)=T(n-b-1) + T(b) + O(n)
- Best case: b=n/2. T(n) = 2T(n/2) + O(n). Which case? T(n) = O(nlogn)
- Worst case: b=n. T(n)=T(n-1) + O(n). T(n) = O(n^2)
- General case: b = n/k, e.g. k=10 => T(n) = O(nlogn)
- Since the pivot is chosen randomly, the probability of the worst case is extremely low. Thus, on average, it is a O(nlogn) algorithm.



# Find the median using Partition (Expected Linear)

- Once we partition the array, we only need to recursively partition the half that contains the median until the index of the pivot is at the median position.
- Time complexity: T(n) = T(n-b) + O(n)
- Best case: b=n/2. T(n)=O(n). Which case?
- Worst case: b=1. T(n) = O(n^2)
- On average, b=n/k, T(n) = O(n)



# Find the median (Worst Case Linear)

- The problem of the previous approach is that there is a tiny probability that the selected pivot point creates unbalanced partitions.
- Worst case linear time complexity can be obtained by always creating balanced partitions.
- We refer kth-smallest for further reading.

# Question 3

3) 12 pts.

Suppose that we have an undirected graph $G = (V, E)$. Prove the following statements.

a) If $G = K_5$, then $G$ is not a planar graph. (3 pts)

b) If $G$ is a simple planar graph with $V < 12$, then $G$ has a vertex of degree 4 or less. (9 pts)

# Important Theorem

In any simple connected planar graph with at least 3 vertices, $E <= 3V-6$

# Question 3

a) If $G$=$K$5, then $G$ is not a planar graph. (3pts)

Answer: In K5, E=10, V=5. E > 3V-6. Contradiction

# Question 3

b) If $G$ is a simple planar graph with $V$<12, then $G$ has a vertex of degree 4 or less.

In order to apply the theorem, we have to make sure V>=3. So

1. If V <= 2, then every vertex has degree at most 1
2. If V >= 3, we obtain E <= 3V - 6. Proof by contradiction: Assume every vertex in G has degree at least 5. Then, we obtain E >= 5V/2. Combine both inequalities, we can

   5V/2 <= E <= 3V-6 =>

   3V - 6 >= 5V / 2 => V >= 12. Contradiction!

# Question 4

Joe owns all $N$ stores along with a straight avenue of length $L$ miles. He knows that for his $i$th store, which is located at $x_i$ miles from the start point of the avenue, can attract all residents whose house is no more than $r_i$ miles from that store. That is, if one's house distance from the start point of the avenue is within $[x_i - r_i, x_i + r_i]$, then he/she will be attracted to Joe's $i$th store. Joe is very proud that no matter where the resident lives in this avenue, he/she will be attracted by at least one of Joe's stores now.

Unfortunately, Joe is short of cash now. He will close some of his stores, but he still wants all the residents in this avenue, no matter where they live, to be attracted by at least one of his stores.

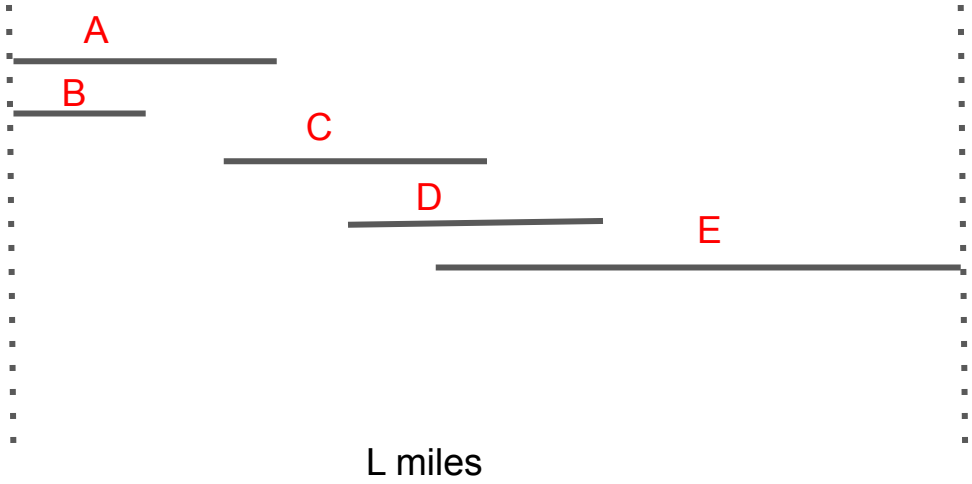
a) Design a greedy algorithm to help Joe to find the minimum stores he can keep but his requirement is still fulfilled. (7 pts)

# Question 4 (Cont')

b) Compute the runtime complexity of your algorithm in terms of $N$. Explain your answer by analyzing the runtime complexity of each step in the algorithm description. (4 pts)

c) Prove the correctness (feasibility and optimality) of your greedy algorithm. (6 pts)

# Analysis




Which ones can be removed?

B and D

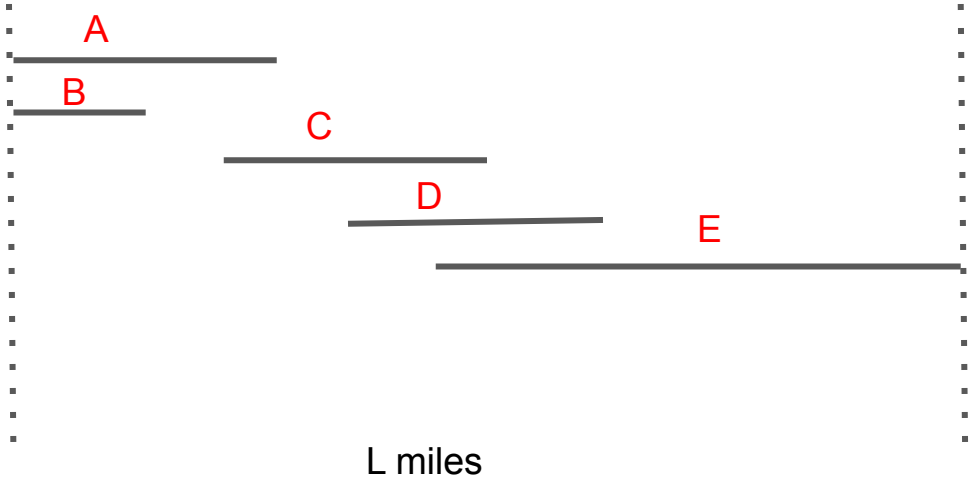
B: Because A fully covers B
D: Because C and E fully covers D

Insights:

1. for any position x, it must be covered by at least one store. Which one do we pick?

Pick the one whose ending distance is maximized.

# Solution




Let the left and right position be L_A and R_A for store A, L_B and R_B for store B, etc.

1. The leftmost uncovered position: 0
   a. A or B?
   b. A
2. The leftmost uncovered position: R_A
   a. Only C
3. The leftmost uncovered position: R_C
   a. D or E?
   b. E
4. The leftmost uncovered position: R_E=L. Done

# Algorithm

1. For $i$th store, we construct the interval $[x_i - r_i, x_i + r_i]$, denoted as $[s_i, f_i]$. The problem now is to find the minimum number of intervals that can cover $[0, L]$.
2. **Sort** all interval by the increasing order of $s_i$ and re-index the stores by this sorting result. (2 pts)
3. Let $S$ be the set of intervals we have chosen and $p$ be the minimum point in $[0, L]$ that intervals in $S$ is not covered. At first, we have $S = \emptyset$ and $p = 0$. Repeat the following steps until $p \geq L$.
    - Iterate the sorted list of intervals and find all intervals $[s_i, f_i]$ such that $s_i \leq p$. Since the list is sorted, we can stop when $s_i > p$. (2 pts)
    - Among the intervals we choose from the first step, we add the interval with the maximum ending point into $S$. If there are multiple candidates, we can choose an arbitrary one. (2 pts)
    - Then, we remove all the intervals that is not chosen in the second step. (1pt)

# Time complexity

- Step 1 will take $O(N)$ time to construct intervals. (Not required)
- Step 2 will take $O(N\log N)$ for sorting. (1pt)
- Step 3: We will visit each interval at most three times: one for finding the intervals covering $p$, one for finding the maximum ending point, and another one for deleting from the list. Therefore, the time complexity of step 3 is $O(N)$. (2pts)
- Thus, the total time complexity is $O(N\log N)$. (1pt)

# Proof

Feasibility: the greedy approach must find a solution

Optimality: the greedy solution finds the one with minimum stores he can keep.

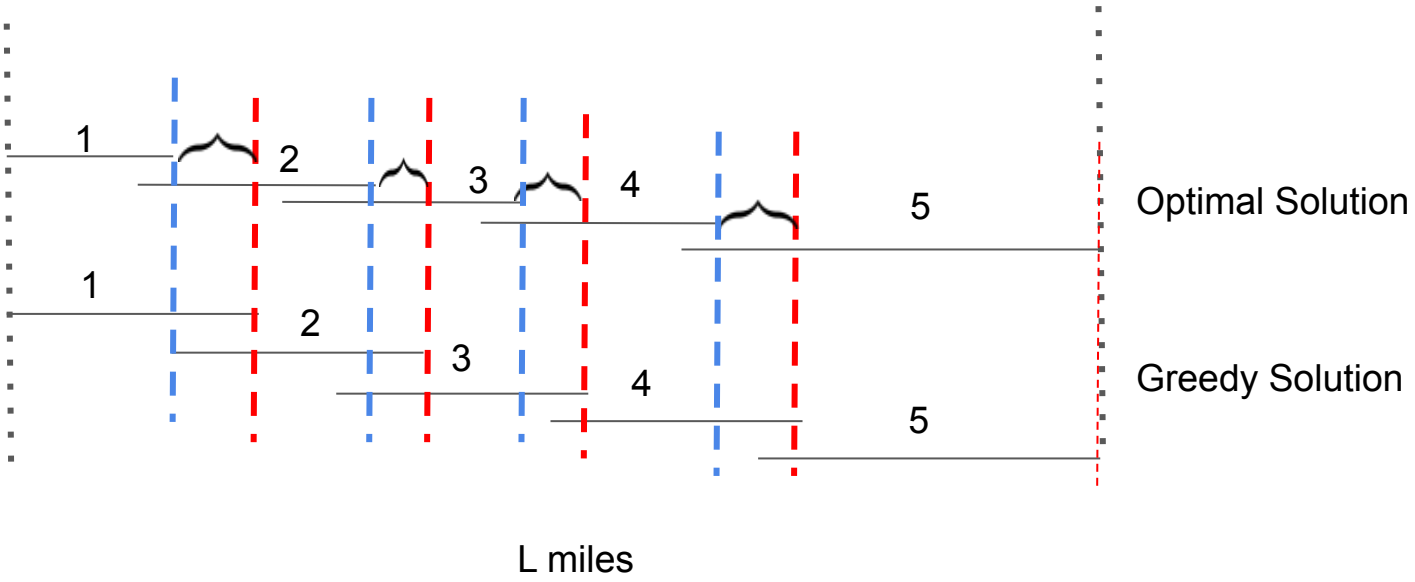
# Feasibility

Prove by contradiction:

The proposed greedy approach selects stores whose coverage is connected. If position x can’t be covered by our approach. That means x can’t be covered by any store in the original list, which is a contradiction.

# Optimality

Core idea: establish a quantity, that the greedy solution is always better than the optimal solution: For the same number of stores, the distance covered by greedy solution is greater than the optimal solution, because we always choose the one that has the maximum ending position.



# Proof

**Optimality:** Suppose that $\{j_1, j_2, \ldots, j_m\}$ is the indices of the intervals in the optimal subset that can cover $[0, L]$. Since it is optimal solution, we have $m \leq k$. We can assume that $i_1 < i_2 < \cdots < i_k$ and $j_1 < j_2 < \cdots < j_m$.

We can prove by induction that $e_{i_p} \geq e_{j_p}$ for all $p \leq m$. (1pt)

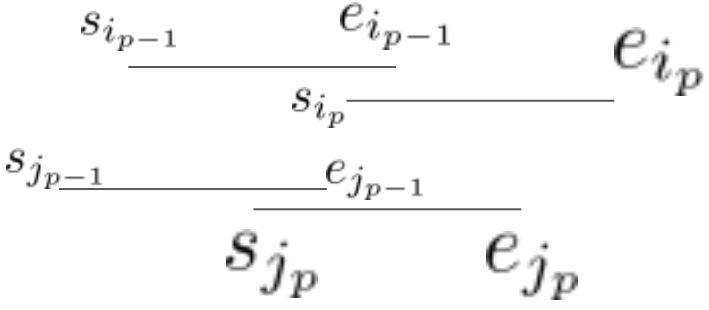
The letter e denotes the ending position

# Proof

**Base case:** $p = 1$. It is true because our greedy algorithm will always choose the one whose ending point is the greatest. (1pt)

**Induction Hypothesis:** If $e_{i_{p-1}} \geq e_{j_{p-1}}$,

**Induction step:** Now we will prove $e_{i_p} \geq e_{j_p}$. We know that to make sure all the intervals have covered $[0, L]$, we must have $e_{i_{p-1}} \geq s_{j_p}$, thus $e_{i_p} \geq s_{j_p}$.Therefore, $[s_{j_p}, e_{j_p}]$ must be considered during the greedy decision of the $p$th interval, then we have $e_{i_p} \geq e_{j_p}$ since we always choose the interval with the maximum ending point. (2pts)




Greedy

Optimal

# Proof

Now, we are going to prove $m = k$ by contradiction. If $m < k$, by the property we have just proved, $e_{j_m} \leq e_{i_m}$. However, by our algorithm, we know that $e_{i_m} < L$. Thus, the optimal subset cannot cover $[0, L]$, which is a contradiction. (1pts)

Thus, our algorithm is the optimal.

# Algorithm thinking & Advanced Heaps

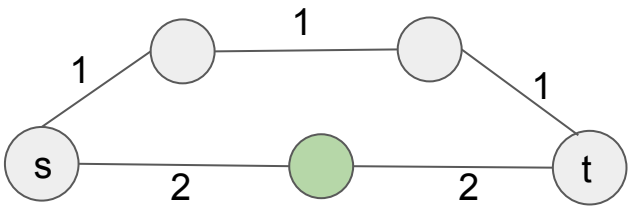
Jiao

# True/False

The shortest path between two points in a graph could change if the weight of each edge is increased by an identical number.

True

Example: Path 1 with edges 1+1+1 = 3. Path 2 with edges 2+2 = 4. Path 1 is shorter. Increase weight of each edge by 2. Path 1 becomes 3+3+3 = 9. Path 2 becomes 4+4 = 8. Path 2 is shorter.



# True/False

Function $f(n) = 5n^2 4^n + 6n^4 3^n$ is $O(n^4 3^n)$

False

$4^n$ dominates $3^n$.

# True/False

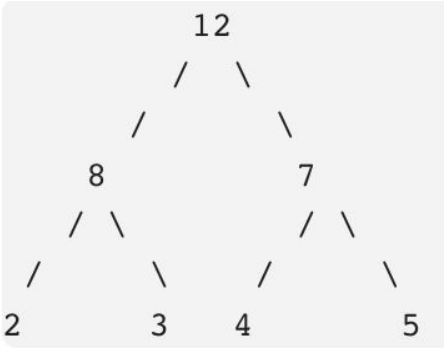
The smallest element in a binary max-heap of size $n$ can be found with at most $n/2$ comparisons.

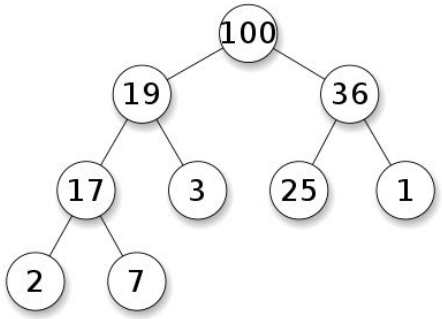
True.

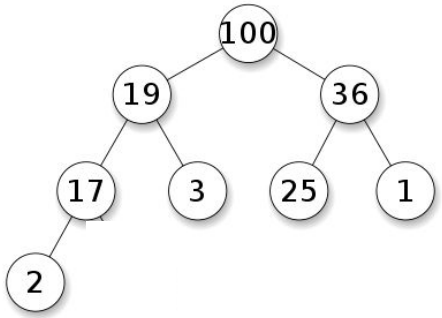
# True/False

The smallest element in a binary max-heap of size $n$ can be found with at most $n/2$ comparisons.

True. In a max heap, **the smallest element is always present at a leaf node.** So we need to check for all leaf nodes for the minimum value -> how many leaf nodes are there?



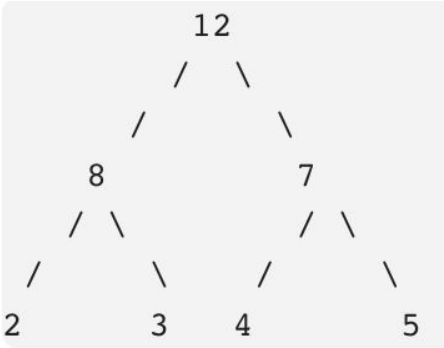



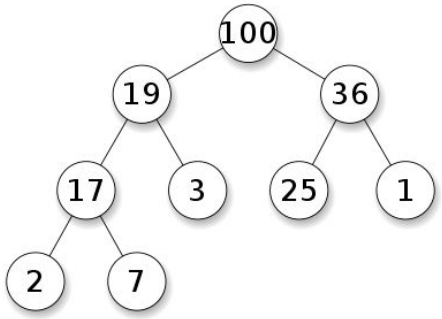



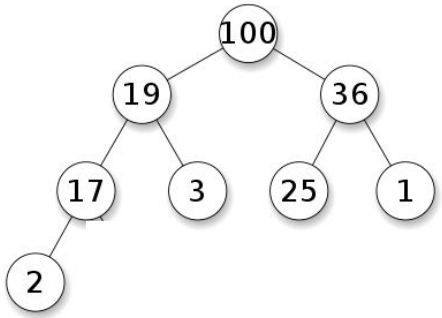
# True/False

The smallest element in a binary max-heap of size $n$ can be found with at most $n/2$ comparisons.

True. In a max heap, **the smallest element is always present at a leaf node.** So we need to check for all leaf nodes for the minimum value -> how many leaf nodes are there? -> **ceil(n/2)** leaf nodes.



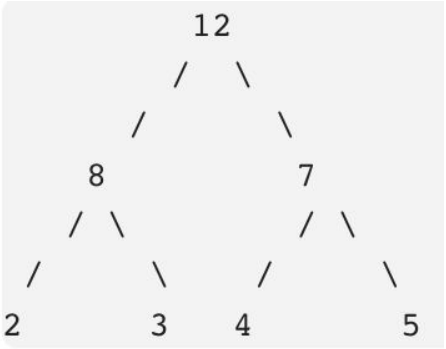



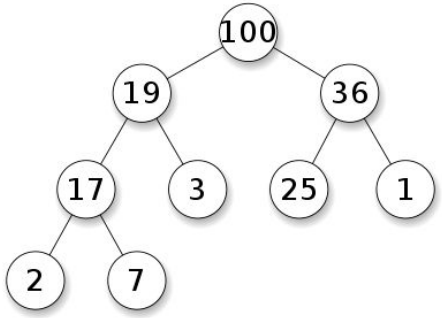



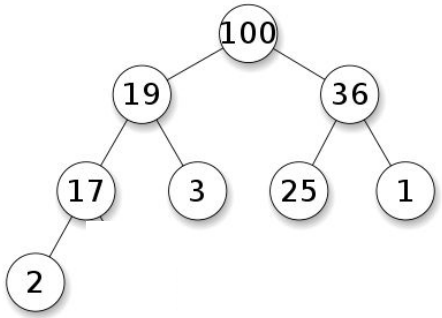
# True/False

The smallest element in a binary max-heap of size $n$ can be found with at most $n/2$ comparisons.

True. In a max heap, **the smallest element is always present at a leaf node.** So we need to check for all leaf nodes for the minimum value -> how many leaf nodes are there? -> **ceil(n/2)** leaf nodes. -> Therefore, we will only have to do at most n/2 comparisons.



# True/False

In a simple, undirected, connected, weighted graph with at least three vertices and unique edge weights, the heaviest edge in the graph is in no minimum spanning tree.

False

If the heaviest edge in the graph is the only edge connecting some vertex to the rest of the graph, then it must be in every minimum spanning tree.

# Question 2

Arrange the following functions in increasing order of growth rate with $g(n)$ following $f(n)$ in your list if and only if $f(n) = O(g(n))$:

1 2 3 4 5 6 7

$$n^{\sqrt{n}}, \quad (\log n)^n, \quad n\,(\log n)^2, \quad n^{\frac{1}{\log n}}, \quad 2^{n^2}, \quad \sum_{i=1}^{n}(i+1)^2, \quad \log(n!)$$

4, 7, 3, 6, 1, 2, 5

Take the log of these functions:

1 $\log\left(n^{\sqrt{n}}\right) = \sqrt{n}\log n$

2 $\log((\log n)^n) = n\log\log n$

3 $\log(n\,(\log n)^2) = \log n + 2\log\log n$

4 $\log\left(n^{\frac{1}{\log n}}\right) = \frac{1}{\log n}\log n = 1$

5 $\log\left(2^{n^2}\right) = n^2\log 2$

6 $\sum_{i=1}^{n}(i+1)^2 = \Theta(n^3) \Rightarrow \log\left(\sum_{i=1}^{n}(i+1)^2\right) = 3\log n$

7 log(log(n!)) = log(log n + log n-1 + log n-2 + … log1) < log(n*logn) = logn + loglogn

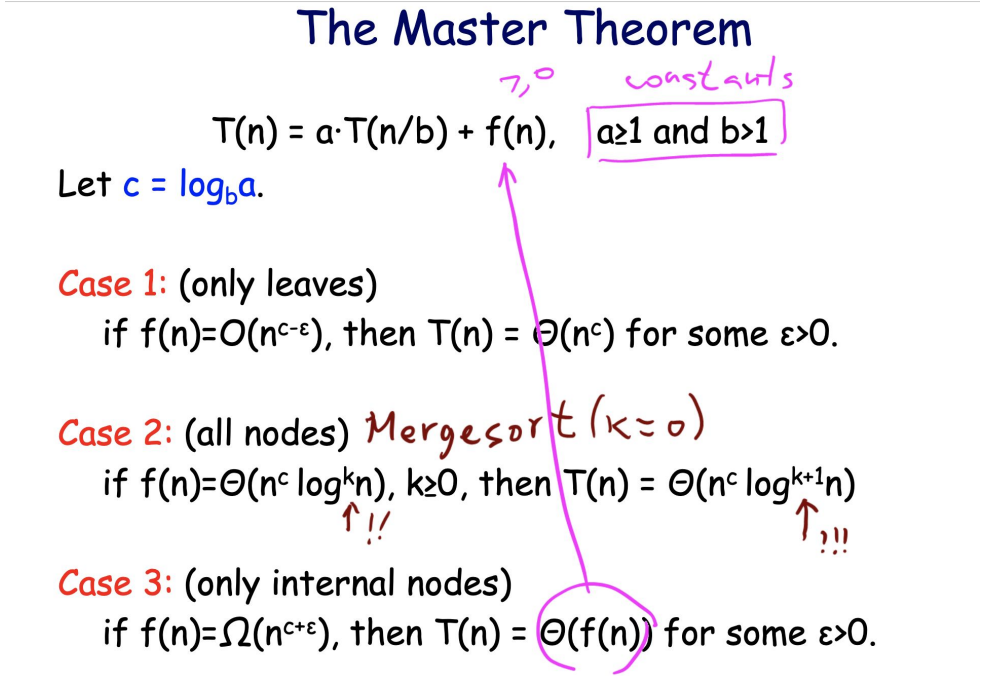
# Question 5

For each of the following recurrences, give an expression in the Theta notation for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

Big-O (**upper** bound): f(n) = O(g(n));
g(n) eventually dominates f(n)

\Omega (**lower** bound): f(n) = \Omega(g(n));
f(n) eventually dominates g(n)

\Theta: f(n) = \Theta(g(n));
f(n) = O(g(n)) **and** f(n) = \Omega(g(n))



# Question 5

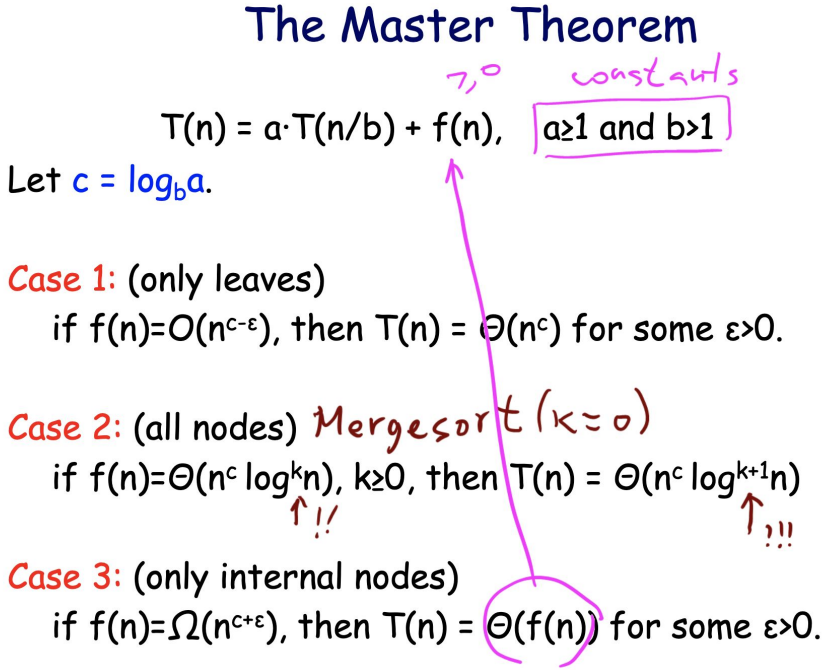
For each of the following recurrences, give an expression in the Theta notation for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

1. $T(n) = 3T\left(\frac{n}{3}\right) + \frac{n}{2}$

a=3, b=3, c=1

case 2 where c=1; k=0

$T(n) = \Theta(n \log n)$



# Question 5

For each of the following recurrences, give an expression in the Theta notation for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

2. $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$

a=2, b=2, c=1;

case 2

why could not apply?

## The Master Theorem

$T(n) = a \cdot T(n/b) + f(n)$, $a \geq 1$ and $b > 1$

Let $c = \log_b a$.

**Case 1:** (only leaves)
if $f(n) = O(n^{c-\varepsilon})$, then $T(n) = \Theta(n^c)$ for some $\varepsilon > 0$.

**Case 2:** (all nodes)
if $f(n) = \Theta(n^c \log^k n)$, $k \geq 0$, then $T(n) = \Theta(n^c \log^{k+1} n)$

**Case 3:** (only internal nodes)
if $f(n) = \Omega(n^{c+\varepsilon})$, then $T(n) = \Theta(f(n))$ for some $\varepsilon > 0$.

# Question 5

For each of the following recurrences, give an expression in the Theta notation for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

3. $T(n) = 16T\left(\frac{n}{4}\right) + n!$

a=16, b=4, c = 2;

Case 3

$T(n) = \Theta(n!)$

## The Master Theorem

$T(n) = a \cdot T(n/b) + f(n)$, $a \geq 1$ and $b > 1$

Let $c = \log_b a$.

**Case 1:** (only leaves)
if $f(n) = O(n^{c-\varepsilon})$, then $T(n) = \Theta(n^c)$ for some $\varepsilon > 0$.

**Case 2:** (all nodes)
if $f(n) = \Theta(n^c \log^k n)$, $k \geq 0$, then $T(n) = \Theta(n^c \log^{k+1} n)$

**Case 3:** (only internal nodes)
if $f(n) = \Omega(n^{c+\varepsilon})$, then $T(n) = \Theta(f(n))$ for some $\varepsilon > 0$.

# Question 5

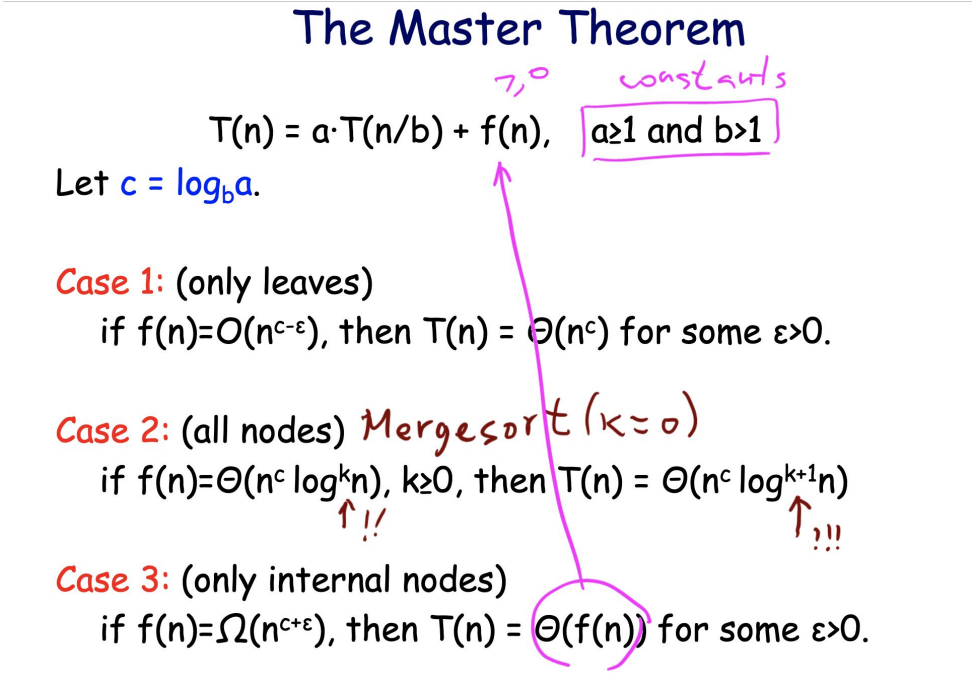
For each of the following recurrences, give an expression in the Theta notation for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

4. $T(n) = 2T\left(\frac{n}{4}\right) + n^{0.51}$

a=2, b=4, c = 0.5;

Case 3

$T(n) = \Theta(n^{0.51})$



# Question 5

For each of the following recurrences, give an expression in the Theta notation for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

$$T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$$

Does not apply.

why?

## The Master Theorem

$T(n) = a \cdot T(n/b) + f(n)$, $a \geq 1$ and $b > 1$

Let $c = \log_b a$.

**Case 1:** (only leaves)
if $f(n) = O(n^{c-\varepsilon})$, then $T(n) = \Theta(n^c)$ for some $\varepsilon > 0$.

**Case 2:** (all nodes)
if $f(n) = \Theta(n^c \log^k n)$, $k \geq 0$, then $T(n) = \Theta(n^c \log^{k+1} n)$

**Case 3:** (only internal nodes)
if $f(n) = \Omega(n^{c+\varepsilon})$, then $T(n) = \Theta(f(n))$ for some $\varepsilon > 0$.

# Minimal Spanning Tree

Zhengfei

# Question 6

T is a spanning tree on an undirected graph G = (V, E). Edge costs in G are NOT guaranteed to be unique. Prove or disprove the following: If every edge in T belongs to SOME minimum cost spanning trees in G, then T is itself a minimum cost spanning tree.

# Question 6

T is a spanning tree on an undirected graph G = (V, E). Edge costs in G are NOT guaranteed to be unique. Prove or disprove the following: If every edge in T belongs to SOME minimum cost spanning trees in G, then T is itself a minimum cost spanning tree.

Prove or Disprove?

# Question 6

T is a spanning tree on an undirected graph G = (V, E). Edge costs in G are NOT guaranteed to be unique. Prove or disprove the following: If every edge in T belongs to SOME minimum cost spanning trees in G, then T is itself a minimum cost spanning tree.

Prove or Disprove?

# Question 6

Counter example:

T = {ab, bc}

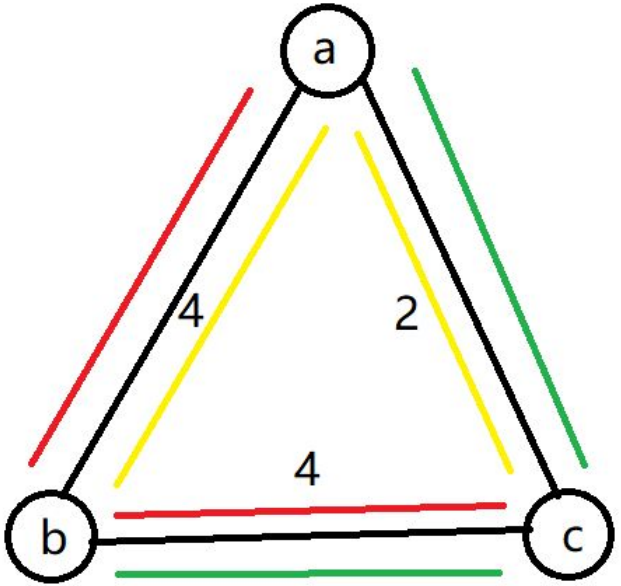
MST1={ab, ac}

MST2={ac, bc}

ab is in MST1

bc is in MST2

However, T is not a MST.



# Divide and Conquer
# Dynamic Programming

# Question 6, HW3

Joseph recently received some strings as his birthday gift from Chris. He is interested in the similarity between those strings. He thinks that two strings $a$ and $b$ are considered J-similar to each other in one of these two cases:

1. $a$ is equal to $b$.
2. he can cut $a$ into two substrings $a_1, a_2$ of the same length, and cut $b$ in the same way, then one of following is correct:
   (a) $a_1$ is J-similar to $b_1$, and $a_2$ is J-similar to $b_2$.
   (b) $a_2$ is J-similar to $b_1$, and $a_1$ is J-similar to $b_2$.

Caution: the second case is not applied to strings of odd length.
He ask you to help him sort this out. Please prove that only strings having the same length can be J-similar to each other, then design an algorithm to determine if two strings are J-similar within $O(nlogn)$ time (where $n$ is the length of strings).

# Naive approach ---- O(n^2)

Apply Divide and Conquer:

Let J(a, b) denotes whether a and b is J-similar

If a == b, return True.

Else, if |a| is odd, return False.

Else, split a and b into a1, b1, a2, b2.

If J(a1, b1)=T and J(a2, b2)=T, return True.

Else, if J(a1, b2)=T and J(a2, b1)=T, return True.

Else, return False.

# Naive approach ---- O(n^2)

Complexity analysis:

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

$\Rightarrow$ $T(n) = O(n^2)$ // The first case of the Master Theorem

# Interesting approach

Luckily, J-similar has the Transitive Property and the Symmetry Property:

Transitive Property: “J(a, b) and J(b, c) ⇒ J(a, c)”

Symmetric Property: “J(a, b) ⇒ J(b, a)”

Using these properties we can speed up the algorithm.

# Interesting approach

When we split a and b into a1, b1, a2, b2:

We first check J(a1, b1) and J(a1, b2).

If J(a1, b1)= F and J(a1, b2)=F, return False.

If J(a1, b1)= T and J(a1, b2)=F, we only need to check J(a2, b2).

If J(a1, b1)= F and J(a1, b2)= T, we only need to check J(a2, b1).

If J(a1, b1)= T and J(a1, b2)= T, here's the interesting part:

Because of the properties above, J(b1, b2) = T.

So, if J(a2, b1) = F, then J(a2, b2) = F. Vise versa.

In both case we only need to check one more time.

# Interesting approach

Complexity analysis:

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

What will be the complexity?

# Interesting approach

Complexity analysis:

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

$\Rightarrow \quad T(n) = O(n^{\log_2(3)}) \sim= O(n^{1.58})$ // The first case of the Master Theorem

Still not O(nlogn) :(

# Standard Solution:

We've already now J(a, b) has the Transitive Property and the Symmetry Property.

We can build two sets Sa and Sb, where Sa = {x | J(a, x)=T}, Sb = {x | J(b, x)=T}.

If J(a, b)=T, then because of those properties, Sa=Sb. (Why?)

Similarly, if J(a, b)=F, then $|Sa \cap Sb| = 0$. (Why?)

# Standard Solution:

Strings can compare with each other just like numbers, by an order called the **Lexicographic Order**

It's like ordering the words in a dictionary: Firstly, compare the first letter of two strings, if they are different, then compare them directly. Otherwise, turn to the second letter and move on. If one of the strings ends during the comparison, then it’s considered as the smaller one.

E.g. “abc” < “acc”, “ab” < “baaaaa”, “CSCI470” < “CSCI570”, “ab” < “abc”

# Standard Solution:

We can sort the elements in Sa and Sb separately.

Then, if Sa=Sb, we have min(Sa)=min(Sb).

If $|Sa \cap Sb|=0$, we have min(Sa) != min(Sb).

By checking if min(Sa)=min(Sb), we can find the value of J(a, b).

How to get min(Sa)?

# Standard Solution:

Divide and Conquer!

For string a, let's say J_min(a) is the minimal string of Sa.

If |a| is odd, then Sa={a}. Obviously J_min(a) = a.

Else, we separate a to a1, a2. Use recursion to get J_min(a1), J_min(a2).

Since Sa = {x+y | x∈Sa1, y∈Sa2 or x∈Sa2, y∈Sa1} ( "+" means concatenation), we have:

$$\min(Sa) = \min\left( \min(Sa1)+\min(Sa2), \min(Sa2)+\min(Sa1) \right)$$

$$\Rightarrow J\_min(a) = \min\left( J\_min(a1)+J\_min(a2), J\_min(a2)+J\_min(a1) \right)$$

# Standard Solution:

Pseudo code (Python style):

```
def J_min(a):
        If |a| is odd:
                Return a
        a1, a2 = split_string(a) # O(n)
        a1_min = J_min(a1)
        a2_min = J_min(a2)
        If a1_min < a2_min: # O(n)
                Return a1_min+a2_min # O(n)
        Else:
                Return a2_min+a1_min # O(n)
```

# Standard Solution:

Complexity Analysis:

Divide and Conquer: T(n) = 2 * T(n/2) + O(n) //concatenation

T(n) = O(nlogn)

Check if J_min(a)=J_min(b): O(n)

Answer: **O(nlogn)**

# Even better Solution:

With the help of hashing, we can actually a way to make the combine step even faster.

Hashing: encode a string to an integer. If two strings are identical then their hash number should be the same.

The recursion formula could be like T(n) = 2*O(n/2) + O(1)

T(n) = **O(n)** !

Theoretically optimal solution.

# Dynamic Programming

Nazanin

# Practice Exam - Q7

Kara is the owner of a fried chicken restaurant called Kara's Fried Chicken. One night, she receives a request for providing lunch for a conference at USC on the next day. The coordinator of the conference has collected lots of orders, but Kara's restaurant is too small to support all of the orders. She has to decide to forfeit some orders, but she wants to earn as much as she can. Here is the information she has now.

- There are $N$ orders from the conference. For the $i$th order, Kara knows it will take $t_i$ minutes to prepare and will need $k_i$ pounds of chicken.
- The coordinator will pay $c_i$ dollars if Kara can deliver the $i$th order. If Kara cannot deliver that order, she could cancel now without paying any cost.
- Kara has only $T$ minutes and $K$ pounds of chicken for the lunch of the conference in the next morning.
- $T, K, N$, and $c_i, t_i, k_i$ for all $i$ are positive integers for simplicity.

Now, Kara has to respond to the coordinator of the conference with which orders she could deliver. Please give an algorithm using dynamic programming to help Kara.

# Understanding the question

- What is the limitation here?
  - Kara has limited time and pounds of chicken for making money. **T** minutes and **K** pounds of chicken. So, she need to use her time and available pounds of chicken wisely.
- What are we looking for?
  - Kara wants to maximize her earning by selecting good orders to spend her total available time and pounds of chicken.
- What are the variables?
  - There are **N** orders for Kara. Each order i:
    - Takes $\mathbf{t_i}$ minutes to prepare
    - Needs $\mathbf{k_i}$ pounds to chicken to be ready
    - Makes $\mathbf{c_i}$ dollars for Kara

# a) Subproblems to be solved

Define **OPT[i,m,p]** as the maximum payment Kara can earn given **m** minutes and **p** pounds of chicken if we consider the first **i** orders.

Then the final answer of the question would be OPT[N, T, K].

# b) Recurrence Relation

Base case (`i = 0, 0 <= m <= T, 0 <= p <= K`):

$$OPT[0, m, p] = 0$$

For recurrence relation of `OPT[i, m, p]` (`i > 0, 0 <= m <= T, 0 <= p <= K`), there are two options:

1. Not selecting order i and getting the maximum earning with this resources (`m` minutes and `p` pounds), from orders 1 to i - 1:

   $$OPT[i - 1, m, p]$$

2. **If** time and resources are <u>enough</u> ($m \geq t_i$ and $p \geq k_i$), there is a option of doing i-th order with the required resources for order i ($t_i$ minutes and $k_i$ pounds) and getting $\mathbf{c_i}$ earning and use the remaining resources to get maximum earning for orders 1 to (i - 1):

   $$OPT[i - 1, m - t_i, p - k_i] + c_i$$

# b) Recurrence Relation

Base case (`i = 0`):

$$OPT[0, m, p] = 0$$

For `i > 0`:

If $m \geq t_i$ and $p \geq k_i$ (there is enough resources to select order i):

$$OPT[i, m, p] = \max(OPT[i - 1, m, p], OPT[i - 1, m - t_i, p - k_i] + c_i)$$

else :

$$OPT[i, m, p] = OPT[i - 1, m, p]$$

---

For example, if order i takes 2 minutes and 3 pounds of chicken to prepare:

available minutes (m = 6):

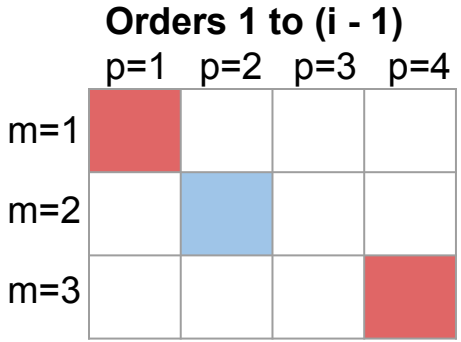
available pounds of chicken (p = 8):

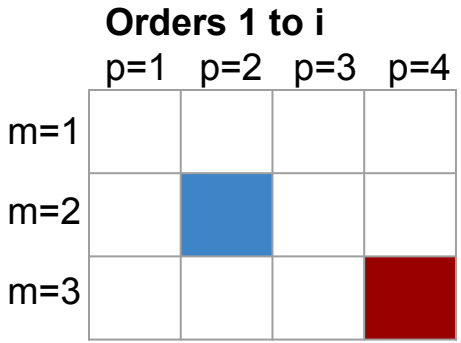
# c) Pseudo code

```
# Base Case (i = 0)
for m = 0 to T:
    for p = 0 to K:
      OPT[0, m, p] = 0

# Recurrence (i > 0)
for i = 1 to N:
    for m = 0 to T:
      for p = 0 to K:
          if t[i] >= m and k[i] >= p: # Resources are enough to make order i
              OPT[i, m, p] = max( OPT[i - 1, m, p], c[i] + OPT[i - 1, m - t[i], p - k[i]] )
          else:
              OPT[i, m, p] = OPT[i - 1, m, p]
```

For example for $t_i = 2$, $k_i = 3$:



# d) Time complexity

There are N*T*K subproblem and each subproblem takes O(1) time to update. So, the total time complexity of the proposed solution is **O(N*T*K)**.

# HW3 - Q3

There are $n$ cities where for each $i < j$, there is a road from city $i$ to city $j$ which takes $T_{i,j}$ time to travel. Two travelers Marco and Polo want to pass through all the cities, in the shortest time possible. In the other words, we want to divide the cities into two sets and assign one set to Marco and assign the other one to Polo such that the sum of the travel time by them is minimized. You can assume they start their travel from the city with smallest index from their set, and they travel cities in the ascending order of index (from smallest index to largest). The time complexity of your algorithm should be $O(n^2)$. Prove your algorithm finds the best answer.

# Subproblems

Let's define **c[i, j]** for each i < j as the minimum possible time to travel through all the cities from i to n, while Marco starts from city **i** and Polo starts from city **j**.

# Recurrence Relation

Base case: C[n-1, n] = 0

Recurrence:

1. if `i < j - 1:`

   $C[i, j] = T_{i, i+1} + T_{i+1, i+2} + T_{i+2, i+3} + \ldots + T_{j-2, j-1} + c[j - 1, j]$

2. if `i = j - 1` then `c[i, j]` is the minimum of all the following cases:
   - Travel city **j - 1** with Marco and give the rest to Polo: $T_{j, j+1} + T_{j+1, j+2} + \ldots + T_{n-1, n}$
   - Travel city **j** with Polo and give the rest to Marco: $T_{j-1, j+1} + T_{j+1, j+2} + \ldots + T_{n-1, n}$
   - Add city **j + 1** to Marco: $\min\{ T_{j - 1, j + 1} + T_{j, k} + c[j + 1, k] \mid j + 1 < k <= n\}$
   - Add city **j + 1** to Polo: $\min\{ T_{j - 1, k} + T_{j, j + 1} + c[j + 1, k] \mid j + 1 < k <= n\}$

# Helper Variable

- We always need sum of traveling to of consecutive cities, starting from city i. We can speed up our recurrence using a helper variable S.
- Let's define $S_j$ as $T_{j, j+1} + T_{j+1, j+2} + \ldots + T_{n-1, n}$
- We can preprocess S, before solving the problem.

$$S_n = 0$$

$$S_j = T_{j, j+1} + S_{j+1}$$

- The time complexity of preprocessing this helper variable is O(n).
    - More details in following slides …

# Recurrence Relation + Helper Variable

Base case: C[n-1, n] = 0

Recurrence:

1. if `i < j - 1:`

   $c[i, j] = T_{i, i+1} + T_{i+1, i+2} + T_{i+2, i+3} + \ldots + T_{j-2, j-1} + c[j - 1, j] = s_i - s_j + c[j - 1, j]$

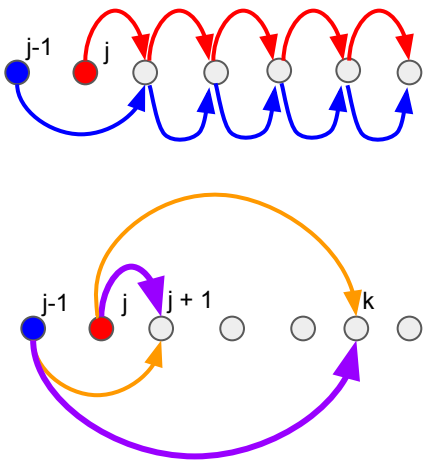
2. if `i = j - 1` then `c[i, j]` is the minimum of all the following cases:
   - Travel city **j - 1** with Marco and give the rest to Polo: $s_j$
   - Travel city **j** with Polo and give the rest to Marco: $T_{j - 1, j + 1} + s_{j + 1}$
   - Add city **j + 1** to Marco: $\min\{ T_{j - 1, j + 1} + T_{j, k} + c[j + 1, k] \mid j + 1 < k <= n\}$
   - Add city **j + 1** to Polo: $\min\{ T_{j - 1, k} + T_{j, j + 1} + c[j + 1, k] \mid j + 1 < k <= n\}$

# Pseudo code

```
s[n] = 0
for i=n-1 to 1:
  s[i] = T[i][i + 1] + s[i + 1]
c[n-1][n] = 0 # Base case
for i=1 to n-2:
    c[i][n] = s[i] - s[n]
for j=n-1 to 2:
  minval = min(s[j], T[j - 1, j + 1] + s[j + 1]) #Case 1 & 2
  for k=j+2 to n-1:
    minval = min(minval, T[j - 1, j + 1] + T[j, k] + c[j + 1][k]) #Case 3
    minval = min(minval, T[j - 1, k] + T[j, j + 1] + c[j + 1][k]) #Case 4

  c[j - 1][j] = minval
  for i=1 to j-2:
    c[i][j] = c[j - 1][j] + s[i] - s[j]

# Marco starts from 1, and Polo starts from j
final_answer = c[1][2]
for j=3 to n:
    final_answer = min(final_answer, c[1][j])
return final_answer
```



# Time complexity

- For i < j - 1 (O(n^2) such subproblems) each update takes O(1).
- For i = j - 1 (O(n) such subproblems) each update takes O(n).
- The pre-processing of array S takes O(n) time.

  ➢ In total time complexity of the algorithm is O(n + n^2 * 1 + n * n) = O(n^2)

# Homework 3 Hints
# Questions 3, 5, 7

Nazanin

# HW3 - Q3

There are $n$ cities where for each $i < j$, there is a road from city $i$ to city $j$ which takes $T_{i,j}$ time to travel. Two travelers Marco and Polo want to pass through all the cities, in the shortest time possible. In the other words, we want to divide the cities into two sets and assign one set to Marco and assign the other one to Polo such that the sum of the travel time by them is minimized. You can assume they start their travel from the city with smallest index from their set, and they travel cities in the ascending order of index (from smallest index to largest). The time complexity of your algorithm should be $O(n^2)$. Prove your algorithm finds the best answer.

# Bad idea!

Define OPT[i] as the best division of cities 1 to i to two sets such that the cost of traveling would be minimum. Then:

$$OPT[i] = \min(OPT[i-1] + T_{max(A),\, i}\ ,\ OPT[i-1] + T_{max(B),\, i}\ )$$

Update A and B based on the minimum value.

Why this is a **bad idea**?

- Because they min of i-1 does not necessary is the min of i.
- Let’s come up with an example to show this is not going to work!

# Bad idea!

Define OPT[i] as the best division of cities 1 to i to two sets such that the cost of traveling would be minimum. Then:

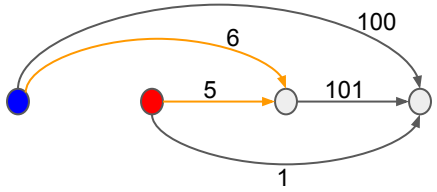
$$OPT[i] = \min(OPT[i-1] + T_{max(A),\, i}\ ,\ OPT[i-1] + T_{max(B),\, i}\ )$$

Update A and B based on the minimum value.

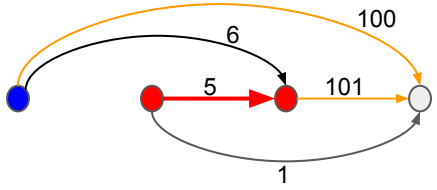
Why this is a **bad idea**?

- Because they min of i-1 does not necessary is the min of i.
- Let's come up with an example to show this is not going to work!

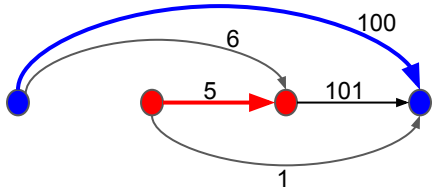
A = [1]
B = [2]




A = [1]
B = [2, 3]

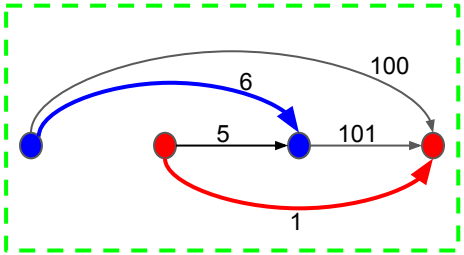



A = [1, 4]
B = [2, 3]



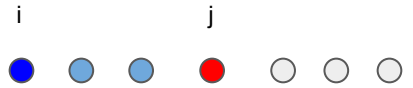

**Well, there is a better solution:**

A = [1, 3]
B = [2, 4]



# Q3 - Hints

1. For i < j, define OPT[i, j] as min cost to cover cities **i to n** while one traveler **starts at city i** and the other **starts at city j**.

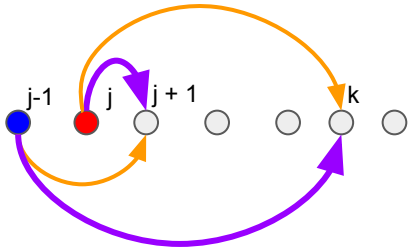



2. If i != j-1, then the first traveler should cover all the cities from i to j - 1. So, for i != j-1


$$OPT[i, j] = T_{i, i+1} + T_{i+1, i+2} + T_{i+2, i+3} + \ldots + T_{j-2, j-1} + OPT[j-1, j]$$

➢ So, I just need to solve problem for OPT[j-1, j] now!

# Q3 - Hints




3. For the recurrence relation, we should consider both orange and purple cases:

4. Make sure to cover this special case where you assign 1 city to one traveler, and remaining n-1 city to the other one.

5. You might need to use the following helper variable to speed up your algorithm:

$$S_1 = 0$$

$$S_i = S_{i-1} + T_{i-1, i}$$

$$T_{i, i+1} + T_{i+1, i+2} + T_{i+2, i+3} + \dots + T_{j-2, j-1} = S_{j-1} - S_i$$

# HW3 - Q5

Due to the pandemic, You decide to stay at home and play a new board game alone. The game consists an array $a$ of $n$ positive integers and a chessman. To begin with, you should put your character in an arbitrary position. In each steps, you gain $a_i$ points, then move your chessman at least $a_i$ positions to the right (that is, $i' >= i + a_i$). The game ends when your chessman moves out of the array.

Design an algorithm that cost at most $O(n)$ time to find the maximum points you can get. Explain your algorithm and analyze its complexity.

# Not an efficient idea!

Let's define OPT[i] as the maximum point I can get from cells 1 to i (ending at cell i). Then use the following recurrence relation:

OPT[0] = 0

OPT[1] = $a_1$

OPT[i] = **max{**OPT[j] + $a_i$ **|** 0 <= j < i, **if** j == 0 or $a_j$ >= (j - i)**}**

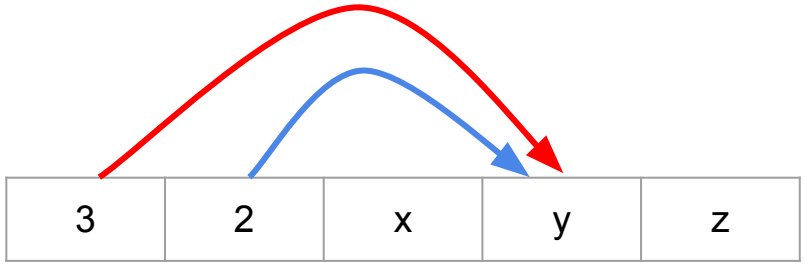
Then the final answer is

max{OPT[i] | 1 <= i <= n}

**Correct**, but **not efficient** (Time complexity: $O(n^2)$)

# How can I make it better?

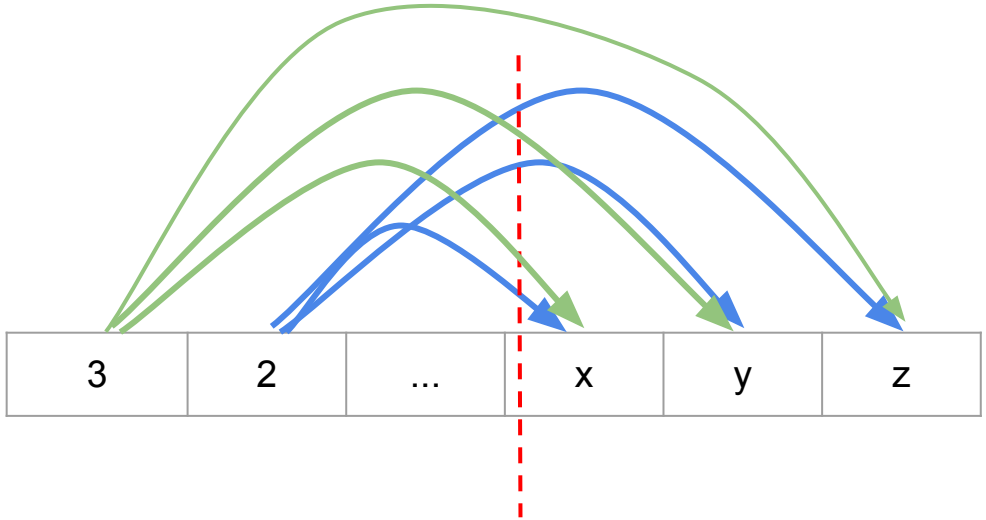
- For having the best answer starting from the first or the second cell, I only care about whether I can reach a particular cell or not (Let's say cell y). **If I can reach cell y**, then cell y plays **the same role** in the update of the first and the second cell.




- So, If I **change the definition of my subproblem** a little bit, then I can have a faster recurrence relation.

# Q5 - Hints

1. Change the definition of the sub-problem I proposed in previous slides a little bit.
2. Use **helper variable** to keep track of "**best place to jump to**" if I can pass the certain cell. For example, if I can pass cell x, then the best place to jump to is always the cell with the maximum profit in set of cells {x, y, z}.



# HW3 - Q7

Chris recently received an array $p$ as his birthday gift from Joseph, whose elements are either 0 or 1. He wants to use it to generate an infinite long super-array. Here is his strategy: each time, he inverts his array by bits, changing all 0 to 1 and all 1 to 0 to get another array, then concatenate the original array and the inverted array together. For example, if the original array is [0, 1, 1, 0], then the inverted array will be [1, 0, 0, 1] and the new array will be [0, 1, 1, 0, 1, 0, 0, 1]. He wonders what the array will look like after he repeat this many many times.

He ask you to help him sort this out. Given the original array $p$ of length $n$ and two indices $a, b$ ($n \ll a \ll b$, $\ll$ means much less than) Design an algorithm to calculate the sum of elements between $a$ and $b$ of the generated infinite array $\hat{p}$, specifically, $\sum_{a \le i \le b} \hat{p}_i$. He also wants you to do it real fast, so make sure your algorithm runs less than $O(b)$ time. Explain your algorithm and analyze its complexity.

# Clarifications

1. An example, let's define p as "110". then

   $S_1$ = **110**001

   $S_2$ = **110001**001110

   $S_3$ = **110001001110**0011....

For example, we can ask for the number of ones between index 8 and 11. The answer is 3 (because: 1100010**<u>0111</u>**00011….).

2. Why I cannot generate the string and count number of ones between a and b?
   - Because the generation takes O(b) time (the length of the final string is around b) and the question asks you to do better than that!

# Q7 - Hints

1. Instead of sum{a to b} use sum{1 to b} - sum{1 to (a - 1)}:

   **1100010011**00011….

   **1100010**011100011….

   1100010**0111**00011….

   Then you only need to calculate sum{1 to x} for a given x.

2. If x is n * $2^k$, you already know the answer of sum{1 to x}.
   a. Hint: you are concatenating the reverse of the string to it (go over the given example).