THERE ARE ARTS FAR DARKER THAN WITCHCRAFT...
Introduction to Algorithms
smbc-comics.com

Figure 1: SMBC Comics by Zach Weinersmith

Definition: Person p is **famous** iff (if and only if) everyone in this classroom knows p, but p does not know anyone else in this classroom.

Definition: A single **query** consists of taking a pair of people (p, q) in the classroom, asking if p knows q, and receiving a response.

Problem Formulation: Determine all the famous people in a classroom with n people, using the minimum possible number of queries.

1. Who in this classroom do you suppose is famous?
2. How would we test our hypothesis to the previous question?
3. If our hypothesis is wrong, does that mean nobody is famous?
4. How can we generalize our test to determine all famous people in the classroom?

A First Attempt:

```
1: For all people in the class $p$:
2:     For all people in the class $q$, where $p \neq q$:
3:         Check if $p$ knows $q$. If so, $p$ is not famous.   } 问了每人两次
4:         Check if $q$ knows $p$. If not, $p$ is not famous.
5:     If $p$ is famous, add $p$ to the list of famous people.
6: Return our list of famous people.
       ↳ contradiction：不可能有两个人同时 famous
```

Follow-up Questions:

1. Is this a good algorithm? not
2. What does it mean for an algorithm to be "good?" efficient
3. How many queries does our algorithm use (exactly)? $2n(n-1)$ → 只问每人一次 → $n(n-1)$
4. How can we improve our algorithm?
5. How many famous people can there be, maximum? 1
6. If p does not know q on line 3, is there any information we can deduce? p is not famous

A Better Attempt:

```
1: Maintain a list of famous candidates, intialized to everyone in the classroom.
2: Take any pair $(p, q)$ from the list (if not possible, go to step 5).
3: Check if $p$ knows $q$. If so, $p$ is not famous. If not, $q$ is not famous.
4: Remove the non-famous person from the list, and return to step 2.
5: There is now one famous candidate $c$.
6: For all people in the class $q$ where $c \neq q$:
7:     Check if $c$ knows $q$. If so, nobody is famous. Return null.
8:     Check if $q$ knows $c$. If not, nobody is famous. Return null.
9: $c$ is famous. Return $c$.
```

repeat until one candidate ⇒ cannot take pair

Followup Questions:

1. How many queries does this algorithm use in the worst case? $(n-1) + 2(n-1) = 3(n-1)$
2. Is this better than our old algorithm? yes
3. Is this **always** better than our old algorithm? no

当 $n \leq 3$ 时, not better

Important Points

1. When writing algorithms, pseudocode is acceptable (in fact, it is recommended). You can also write in code, or in English, as long as you provide enough detail. (包括 edge case 等)
2. If a reasonably competent programmer could take your answer and code it up in a language of their choice, then your answer is acceptable.
3. Algorithms is learned with **practice**. If you think that you must have a "Eureka" moment to answer an Algorithms problem on a test, that is merely a sign that you need to practice more problems.

Core Course Questions

1. Given a problem, how do we produce an algorithm to solve it?
2. Given a problem and an algorithm, can we prove that the algorithm correctly solves the problem?
3. Given an algorithm, will it terminate in a reasonable amount of time? polynomial → reasonable
4. What is a reasonable amount of time anyway?
5. Are there problems which cannot be solved in a reasonable amount of time?
6. How would we identify such problems?
7. Are there problems that no algorithm can solve ?

Extra Problems

1. Improve the algorithm for the Famous Person Problem to require only $3(n-1) - \underline{\log n}$ queries.
2. Informally argue why the Famous Person Problem cannot be solved in less than $\theta(n)$ queries.

Proofs and Recurrence Relations

Definition: Given two sets A and B, the Cartesian Product $A \times B = \{(x, y) \mid x \in A, y \in B\}$

Proof Types

1. Proof by Contradiction: if n^2 is odd, then n is odd.

   Assume n is even and n^2 is odd

   $n = 2k$

   $n^2 = 4k^2 = 2(2k^2)$ even $\Rightarrow\Leftarrow$ contradiction

2. Proof by Contradiction: There are an infinite number of primes.

   Assume finite of primes $\Rightarrow$ there is a largest prime p

   fundamental theorem of arithmetic : $p!$ divisible by all the primes

   $\Rightarrow$ $p!+1$ is not divisible by any number between 1 and p

   So its prime factorization contains primes $>p$

3. Prove or disprove: ✓ for any sets A, B, C, if $A \times C = B \times C$, then $A = B$.

Proof Attempt for 3:

1: Assume the opposite: $A \times C = B \times C$ but $A \neq B$.

2: Since $A \neq B$, there must be some element in one set which is not in the other set.

without lost of generality

3: wlog, assume $a \in A$ and $a \notin B$.

4: Take an arbitrary element $c \in C$. we assume $\exists$ c in C

5: By definition of cartesian product, $(a, c) \in A \times C$, and $(a, c) \notin B \times C$. Contradiction with **1**.

case 2 : $C = \phi$

$A \times \phi = \phi$

$B \times \phi = \phi$

6: If $A \times C = B \times C$ then $A = B$.

Are there any holes in this proof?

Proof Tips

- Run through some examples. This will help convince yourself the claim is true, as well as give you an intuitive understanding for **why** its true.

- Use the definition to translate a statement into mathematical form when possible. This makes the proof easier.

- When doing a proof by contradiction, make sure you are assuming the logical opposite. Make a truth table if you have to.

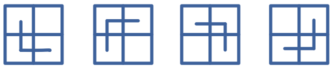
- Finding a proof is not a straight line from A to B. Even the most experienced research scientists go off on incorrect tangents when looking for a proof of a fact. Just keep deriving stuff until you get what you need.

- If you don't know whether you need to prove or disprove a statement, follow your intuition. If you fail to reach the required conclusion, you probably learned something about the problem. Use this new information and try the other path!

Inductive Proofs

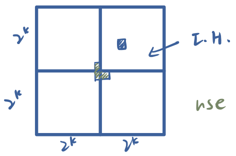
- Prove: Any 2^n x 2^n chessboard with one square removed can be tiled using 3-square L-shaped pieces.

  Base Case :

  I.H. Assume claim is true $\forall n \le k$

  I.S. $n = k+1$

  use I.H. for the other 3 pieces

- Find the flaw in the proof that $a^n = 1$ for all non-negative integers n and all non-zero reals a.
  Base Case: $a^0 = 1$.
  Inductive Assumption: $a^n = 1$, for all $n \le X$.
  Inductive Step: $a^{X+1} = \frac{a^X a^X}{a^{X-1}} = \frac{1 \cdot 1}{1} = 1$.

  ↳ using $a^{X-1}=1$ and $a^X=1$ ⇒ need 2 base cases

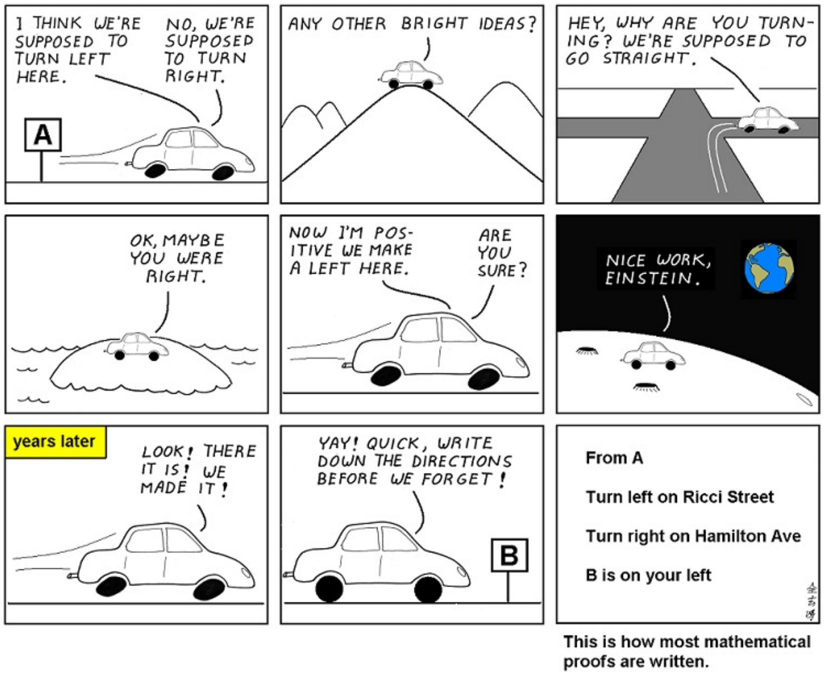
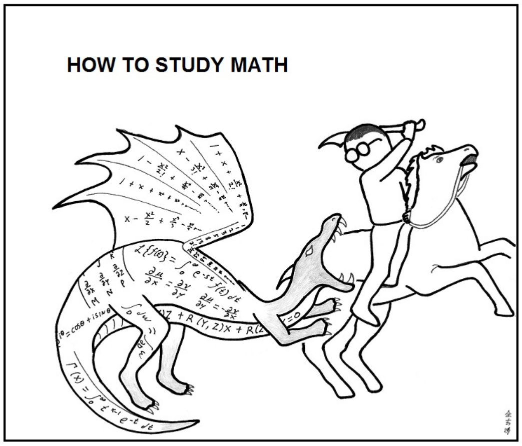

I THINK WE'RE SUPPOSED TO TURN LEFT HERE.
NO, WE'RE SUPPOSED TO TURN RIGHT.
A
ANY OTHER BRIGHT IDEAS?
HEY, WHY ARE YOU TURNING? WE'RE SUPPOSED TO GO STRAIGHT.
OK, MAYBE YOU WERE RIGHT.
NOW I'M POSITIVE WE MAKE A LEFT HERE.
ARE YOU SURE?
NICE WORK, EINSTEIN.
years later
LOOK! THERE IT IS! WE MADE IT!
YAY! QUICK, WRITE DOWN THE DIRECTIONS BEFORE WE FORGET!
B
From A
Turn left on Ricci Street
Turn right on Hamilton Ave
B is on your left
This is how most mathematical proofs are written.

Figure 1: Abstruse Goose # 230: After working on this proof for years, I have finally decided that it IS, in fact, obvious.



Don't just read it; fight it!

--- Paul R. Halmos

Figure 2: Abstruse Goose # 353

```
Mergesort(Array A[1:n])                                  f(n) = 2 f(n/2) + Θ(n)
  If n==1 Then return A      Θ(1)
  B = Mergesort(A[1:n/2])      f(n/2)
  C = Mergesort(A[n/2+1:n])    f(n/2)
  return Merge(B,C)     Θ(n)
```

Questions:

- What is the running time of MergeSort? f(n) = runtime of MergeSort on things
  $\Theta(n \log n)$
- How do you know what the running time is?

A **recurrence relation** is a recursive definition for a sequence of numbers. They can be used to model the running time of a recursive algorithm.

Recurrence Relation for MergeSort: $g(n) = 2g(\frac{n}{2}) + dn$. $g(2) = e$.

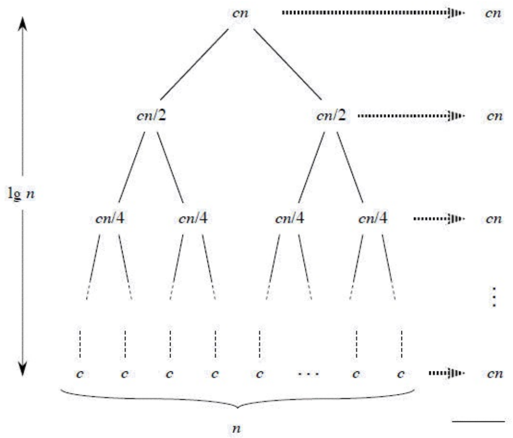
Inductive Proof that MergeSort $= O(n \log n)$

1: Hypothesize that $g(n) \leq cn \log n$, for all $n \geq 2$. Note that we get to choose what c is.
2: Base Case: We need $g(2) = e \leq 2c \log 2 = 2c$. Choose c large enough so that $e \leq 2c$.
3: I.H.: Assume that $g(n) \leq cn \log n$, for all $n : 2 \leq n \leq k$.
4: Consider $g(k+1)$. By the definition, $g(k+1) = 2g(\frac{k+1}{2}) + d(k+1)$.
5: By the inductive hypothesis, $g(\frac{k+1}{2}) \leq c\frac{k+1}{2} \log(\frac{k+1}{2})$.
6: $g(k+1) \leq c(k+1)\log(\frac{k+1}{2}) + d(k+1) = c(k+1)([\log(k+1)] - 1) + d(k+1) = c(k+1)\log(k+1) + (d-c)(k+1)$.
7: We need $c(k+1)\log(k+1) + (d-c)(k+1) \leq c(k+1)\log(k+1)$. Choose c large enough so that $d \leq c$.

$\exists$ such c $\Rightarrow$ proved

Followup Questions:

- Does this prove Mergesort $= \theta(n \log n)$?
- Is it valid to use $n = 2$ as the base case?
- What would have happened if we tried $n = 1$ as our base case?
- What are the disadvantages to solving recurrence relations with Induction?



$$\sum_{i=0}^{\log_2 n} 2^i \cdot \frac{n}{2^i} = \theta(n \log n)$$

Figure 3: A Recursion Tree argument that MergeSort $= \theta(n \log n)$

Master Theorem

- Can solve (almost) any recurrence relation of the form $f(n) = af(\frac{n}{b}) + g(n)$, with constants $a \geq 1, b > 1$.
- Compare $g(n)$ with $n^{\frac{\log a}{\log b}}$.
- Case 1: If $g(n) = \theta(n^{\frac{\log a}{\log b}})$, then $f(n) = \theta(g(n) \log n)$. runtime
- Case 2: If $g(n) = \Omega(n^{\frac{\log a}{\log b}+\epsilon})$, for some $\epsilon > 0$, then $f(n) = \theta(g(n))$.
  → bigger at least some bit, not just lower bound
- What should Case 3 be?
  case 3: if $g(n) = O(n^{\frac{\log a}{\log b}-\epsilon})$ for some $\epsilon > 0$, then $f(n) = \theta(n^{\frac{\log a}{\log b}})$

Extra Problems

1. Prove for any integer n: n is odd if and only if $3n + 1$ is even.
2. Use Induction to prove that $\sum_{i=0}^{n} \frac{1}{2^i} < 2$.
3. Chapter 2, exercises 3,4,5,6.
4. Challenge problems: Chapter 2, exercise 8

Advanced Runtime Analysis

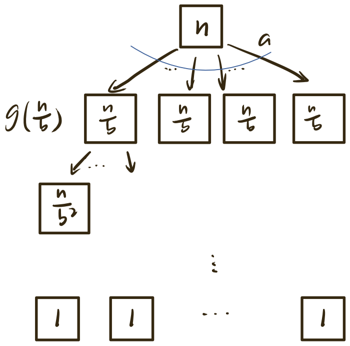
Master Theorem

- Can solve (almost) any recurrence relation of the form $f(n) = af(\frac{n}{b}) + g(n)$, with constants $a \geq 1, b > 1$.
- Compare $g(n)$ with $n^{\frac{\log a}{\log b}}$.
- Case 1: If $g(n) = \theta(n^{\frac{\log a}{\log b}})$, then $f(n) = \theta(g(n) \log n)$. runtime
- Case 2: If $g(n) = \Omega(n^{\frac{\log a}{\log b}\underline{+}\epsilon})$, for some $\epsilon > 0$, then $f(n) = \theta(g(n))$.
  → bigger at least some bit, not just lower bound
- What should Case 3 be?
  case 3: if $g(n) = O(n^{\frac{\log a}{\log b} - \epsilon})$ for some $\epsilon > 0$, then $f(n) = \theta(n^{\frac{\log a}{\log b}})$

Master Theorem Questions

- How much work is done at the top level?
- How many children does the root node have?
- How much work is required at a child node?
- How many grandchildren are there?
- How much work at a grandchild node?
- What is the depth of the tree?
- How many leaf nodes are there?
- How much work at a leaf node?
- How much work is done at the bottom level?
- Solve: $f(n) = 2f(\frac{n}{2}) + cn$
- Solve: $f(n) = f(\frac{n}{2}) + 1$
- Solve: $f(n) = 8f(\frac{n}{2}) + 1000n^2$

推导 by tree:



depth: $\log_b n$

$\Rightarrow a\, g(\frac{n}{b})$

$\Rightarrow a^2 g(\frac{n}{b^2})$

$\Rightarrow a^{\log_b n} \cdot \Theta(1)$

$= n^{\log_b a} \cdot \Theta(1)$ (rule of logarithm)

$= n^{\frac{\log a}{\log b}} \cdot \Theta(1)$

master theorem: compare top level with bottom level

如果 $g(n)$ 大，只看 $g(n)$

如果 $n^{\frac{\log a}{\log b}}$ 大，只看 $n^{\frac{\log a}{\log b}}$

如果一样大，那 level · $g(n) = \log_b n \cdot g(n)$

$= g(n) \log n$

use master theorem for mergesort:

$f(n) = 2f(\frac{n}{2}) + \Theta(n)$

$a=2$, $b=2$, $g(n) = \Theta(n)$

$n^{\frac{\log a}{\log b}} = n = \Theta(n)$ $\Rightarrow$ case 1

$\therefore f(n) = \Theta(n \log n)$

use master theorem for binary search:

$f(n) = f(\frac{n}{2}) + \Theta(1)$

$a=1$ $b=2$ $g(n) = \Theta(1)$

$n^{\frac{\log a}{\log b}} = 1 = \Theta(1)$

$\therefore f(n) = \Theta(\log n)$

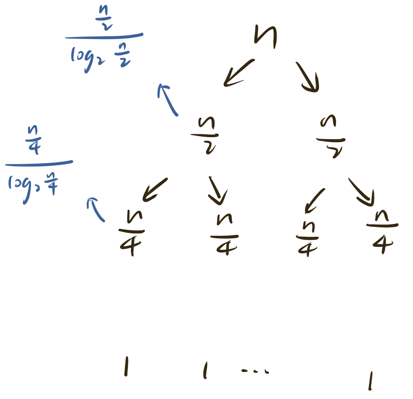
$f(n) = 8(\frac{n}{2}) + 1000n^2$

$a=8 \quad b=2 \quad g(n) = 1000n^2$

$n^{\frac{\log a}{\log b}} = n^3 > 1000n^2$

$\therefore f(n) = \Theta(n^3)$

depth: $\log_2 n$



$\frac{n}{\boxed{\log_2 n}}$ i

$\frac{n}{\log_2 \frac{n}{2}} = \frac{n}{\log_2 n - 1}$

$\frac{n}{\log_2 \frac{n}{4}} = \frac{n}{\log_2 n - 2}$

$\frac{n}{\log_2 n - (\log_2 n - 1)} = \frac{n}{1}$

$\sum_{i=1}^{\log n} \frac{n}{i} = n \sum_{i=1}^{\log n} \frac{1}{i} = \Theta(n \log(\log n))$

↓ 指代分母

$\sum_{i=}^{\log n} \frac{n}{\log_? n} = \Theta($

- Solve: $f(n) = 2f(\frac{n}{2}) + n^2$ $\Theta(n^2)$

  $a=2$ $b=2$ $g(n)=n^2 > n$

- Solve: $f(n) = 2f(\frac{n}{2}) + \frac{n}{\log n}$ (This one is trickier than it looks!)

  $a=2$ $b=2$ $g(n)=\frac{n}{\log n}$

  $n^{\frac{\log a}{\log b}} = n$

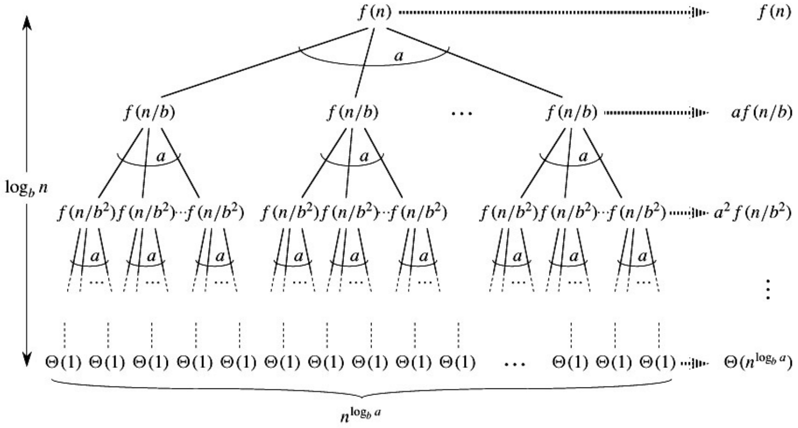
  test case 3: $g(n) = O(n^{\frac{\log a}{\log b} - \epsilon})$ fr some $\varepsilon > 0$ false : 不满足 case 3

  判断是否为 true 不可用 master theorem

  $\frac{n}{\log n} \overset{?}{=} O(n^{1-\epsilon}) = O(\frac{n^1}{n^\epsilon})$ × 要用 tree

  $n^\epsilon \overset{?}{=} O(\log n)$ ×

Analyzing Master Theorem by Recursion Tree (credit: GeeksForGeeks)


f(n)
f(n)
a
f(n/b)
f(n/b)
…
f(n/b)
af(n/b)
a
a
a
log_b n
f(n/b²) f(n/b²) ⋯ f(n/b²)
f(n/b²) f(n/b²) ⋯ f(n/b²)
f(n/b²) f(n/b²) ⋯ f(n/b²)
a²f(n/b²)
a
⋮
Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) … Θ(1) Θ(1) Θ(1)
Θ(n^{log_b a})
n^{log_b a}

Amortized Runtime

Recall the List ADT: (implemented with an array)

push-back analysis

1. insert (int position, T value) best : $\Theta(1)$ worst : $\Theta(n)$ average : $\Theta(1)$

   amortized worst case : $\Theta(1)$

   worst case average case

2. remove(int position)
3. set(int position, T value)
4. T get (int position)

1

2^0

2^1

1

2^2

1

⋮

最后 size 为 n

$$\sum_{i=0}^{\log_2 n} 2^i + \sum_{i=0}^{n-\log_2 n} 1 = n + n - \log_2 n = \Theta(n)$$

$\Rightarrow$ amortized : $\frac{\Theta(n)}{n} = \Theta(1)$

$1 + \cdots + n = \Theta(n)$

Analyze the runtime analysis for each of these operations when implementing the List with an Array.

1. What would be the runtime of Get?

2. What would be the runtime of Remove?

3. What would be the runtime of Insert?

Amortized analysis takes the big picture and says: the first x operations will take no more than $\Theta(y)$ time, for an average of $\Theta(\frac{y}{x})$ per operation. Amortized runtime is the **worst-case average-case**.

- What would be the amortized runtime of Inserting at the end of an ArrayList?

Let's say we have a boolean array as a "counter". Each index starts at 0 (false), and then the counter starts counting up in binary.

Flipping an index from 0 to 1, or from 1 to 0, costs an operation. Counting from 0000 to 0001 only takes 1 operation, but counting from 0011 to 0100 takes 3 operations.

Our increment function should correctly increase the binary number by 1, flipping all necessary bits.

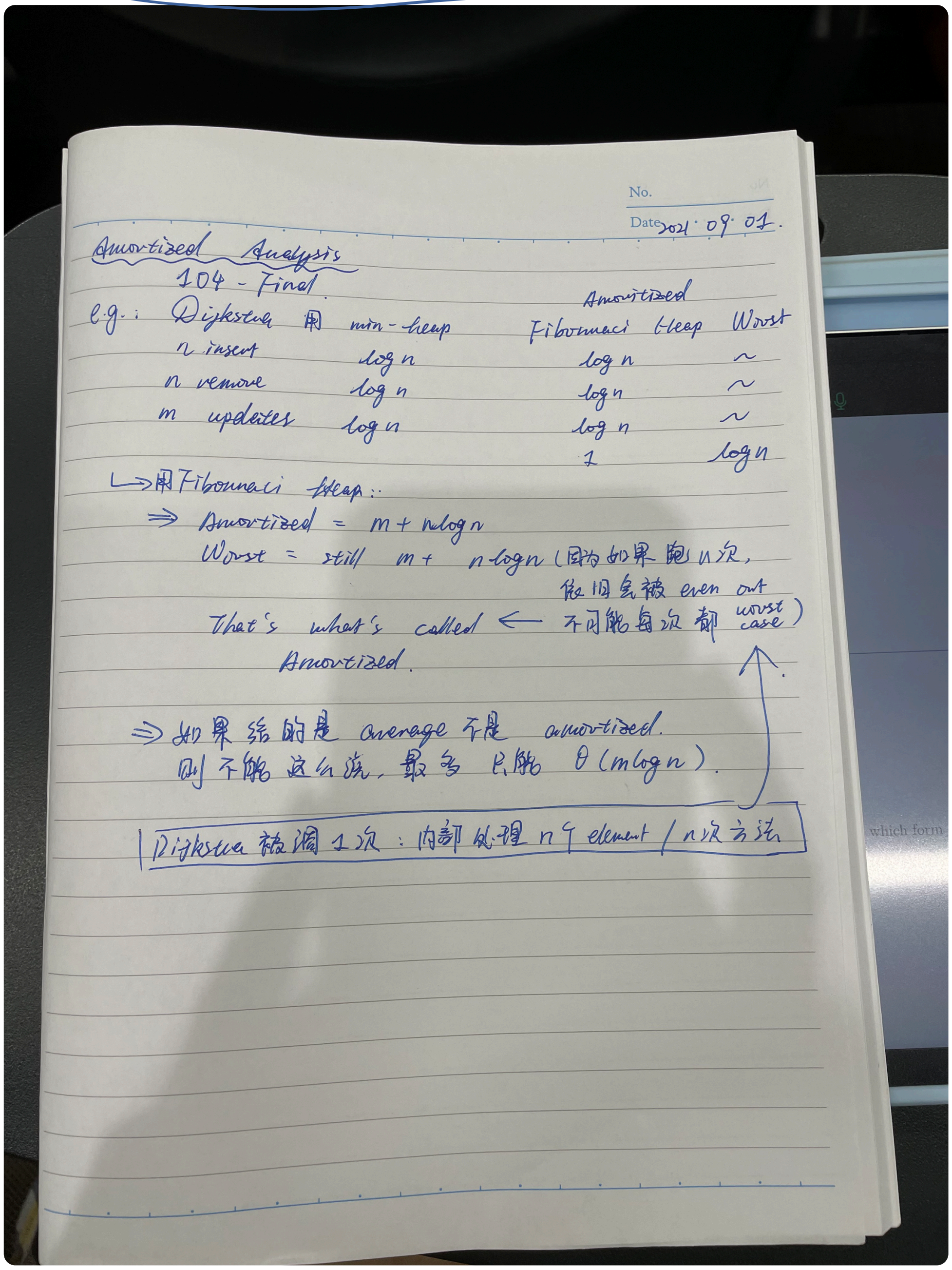
- What is the worst-case runtime of our increment function?
- What is the amortized runtime of our increment function?

Increment	Time	Total Time	Average Time
1	1	1	1
2	2	3	1.5
3	1	4	1.33
4	3	7	1.75
5	1	8	1.6
6	2	10	1.67
7	1	11	1.57
8	4	15	1.88
16	5	31	1.94
32	6	63	1.97
64	7	127	1.98

Fibonacci Heap:

	worst	amortized
peek	O(logn)	Θ(logn)
removeMin	O(logn)	Θ(logn)
inset(n)	O(logn)	Θ(logn
update	O(logn)	Θ(1)

Worst: ~~mnlogn~~ ~~m+nlogn~~
amortized: m+nlogn



No.
Date 2021·09·01.

Amortized Analysis
104 - Final.

e.g.: Dijkstra 用	min-heap	Amortized Fibonacci Heap	Worst
n insert	log n	log n	~
n remove	log n	log n	~
m updates	log n	log n	~
		1	log n

↳用Fibonacci Heap:

⇒ Amortized = m + nlogn
Worst = still m + nlogn (因为如果跑n次，依旧会被 even out 不可能每次都 worst case)

That's what's called ← Amortized.

⇒ 如果给的是 average 不是 amortized.
则不能这么说，最多只能 Θ(mlogn).

Dijkstra 被调1次：内部处理n个 element / n次方法

Extra Problems

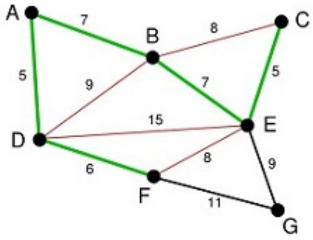
1. Solve the recurrence relation: $f(n) = f(\frac{n}{4}) + \sqrt{n}$
2. Solve the recurrence relation: $f(n) = f(\frac{n}{4}) + 1$
3. Solve the recurrence relation: $f(n) = f(\frac{n}{4}) + \log n$

看 Recording 吧

Minimum Spanning Trees

Given a connected graph $G = (V, E)$, a **Spanning Tree** is a subset of the edges which form a tree on the original nodes.

Given a weighted connected graph, a **Minimum Spanning Tree** (or MST) is the spanning tree which minimizes the sum of the edge weights.



Kruskal's Algorithm: Add edges from smallest to largest, unless an edge creates a cycle.

Reverse-Delete: Remove edges in descending order, unless an edge disconnects the graph.

Prim's Algorithm: Start w/ some root, greedily grow tree like dijkstra's by choosing the smallest edge with exactly one endpoint in the explored set.

m 边
n 点

? What is the runtime of Prim's algorithm? $\theta(m \log n)$

- What would be the first step in the implementation of Kruskal's Algorithm? sort the edges
- What would the runtime of this step be? $\theta(m \log m)$
- What is the runtime of Kruskal's algorithm overall? $\theta(m^2)$

for (each edge e) {
    if adding e doesn't create a cycle, add it ↙ DFS/BFS $\theta(m+n)$
}

$m \cdot \theta(m+n)$
$= \theta(m^2)$

New data structure: Union-Find. Maintain one set for each connected component, consisting of all nodes in that component.

Union(x,y) combines the sets x and y into a single set.

Find(x) determines which set contains node x. If Find(x) =Find(y), you don't want to add the edge (x, y).

```
Kruskal(V,E)
Sort $E$ in ascending order of weight.    $\theta(m\log m)$
For each edge $(x, y)$ in ascending order of weight {
    $a =$ Find$(x)$    which component is x in
    $b =$ Find$(y)$
    If $a \neq b$ Then {
        Union$(a, b)$
        Add edge $(x, y)$
}   }
```

every node 存自己是哪个union:
Find: $\theta(1)$
Union: $\theta(n)$

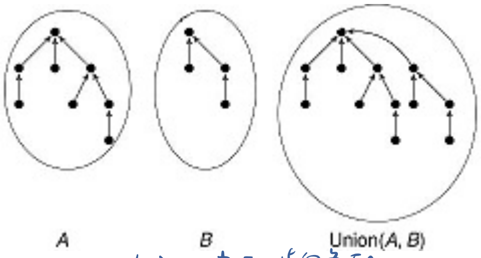
$2m \cdot \theta(1) + (n-1) \cdot \theta(n)$
$= \theta(m + n^2)$
$= \theta(n^2)$

$\Rightarrow$ 总: $\theta(m\log m + n^2)$

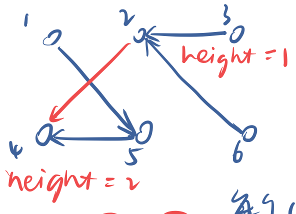
Attempt 1: Each node has a variable which stores the name of the set which it is in.

- How long would Find take? $\Theta(1)$
- How long would Union take? $\Theta(n)$

Attempt 2: Each set is identified by a specific node, whom we'll refer to as the "captain" of the set. Each node has a pointer to another node in the set. The captain's pointer will be null. captain determines the name of a set

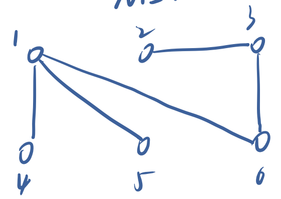


Union Find



∴ ②→④

MST



每个 components track 前 height

height 少的 指向多的

只有连2个 height 相等的 captain 时, height 才增加

just one points to another

n=	height
1	0
2	1
3	1
4	≤2
7	≤2
8	≤3
16	≤4
32	≤5

- How long would Union take, assuming we pass in the captains of sets? $\Theta(1)$
- How long would Find take? worst: $\Theta(n)$ ⇒ $\Theta(\log n)$
- The worst-case analysis of Find requires a rather stupid decision. How could we improve our Union-Find data structure?
- If we do a union between trees of depth x and y, where $x < y$, what is the depth of the new tree? y
- When would a union produce a tree of greater depth? 2 captain with equal height
- What is the depth of a 1,2, or 3 node tree?
- What is the greatest depth possible for a 4 node tree?
- How many nodes are required to produce a depth-3 tree?
- How many nodes are required to produce a depth-4 tree?
- What is the runtime of Find? $\Theta(\log n)$
- What is the runtime of Union, assuming we pass in set captains? $\Theta(1)$
- Putting it all together: what is the runtime of Kruskal's Algorithm? $\overset{\text{sort}}{\Theta(m\log m)} + 2m \cdot \Theta(\log n) + (n-1)\Theta(1)$
- Why does the book give a different runtime analysis for Kruskal's algorithm? $\Theta(m \log n)$

$\because m < n^2$

$\Theta(m\log m) < \Theta(m\log n^2) = \Theta(2m\log n) = \Theta(m\log n)$

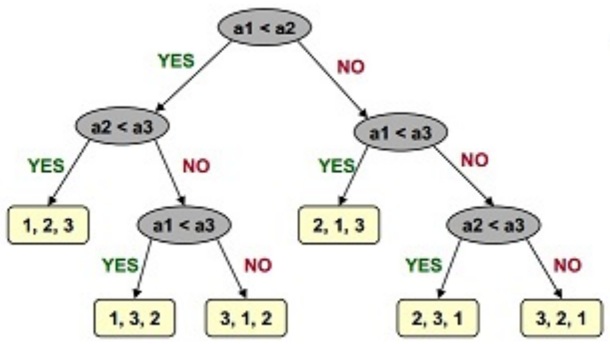
可在每一次 find 时缩减树的高度（找完改成直接指向 captain） ⇒ 减少 union-find 中 runtime

To get a better runtime for Kruskal's, we would need to find a better sorting algorithm. MergeSort obtains $\theta(n \log n)$: is it possible to beat this?

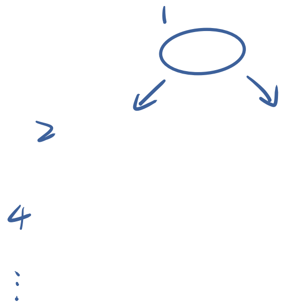
Claim: Comparison-based sorting takes $\Omega(n \log n)$ time.

A sorting algorithm is **comparison-based** if we make comparisons between elements in the array-to-be-sorted, and determine the sorted order based on those comparisons.

Comparison Based Sorting Lower Bound



Decision Tree of Program



- When we reach a leaf node in a decision tree, what do we know?

  It is the final sorted order

- How many leaf nodes must there be when we wish to sort n numbers?

  $\geq n!$ (每个 leaf node 指向一种 permutation)

- How deep must the tree be to achieve this?

  $\geq \underline{\log n!}$

  ↳最底层是 $n!$

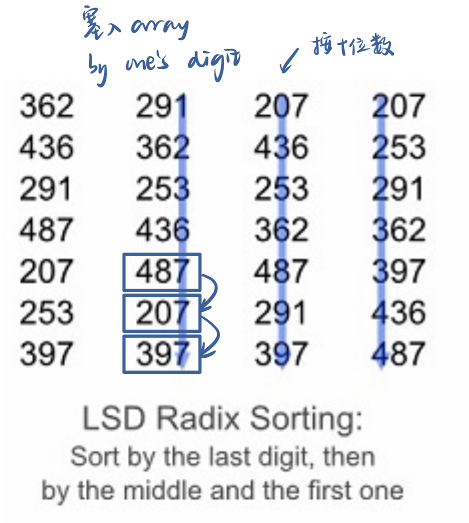
  $n! \neq \Theta(n^n)$

  $n! = O(n^n)$

  $\log n! = \Theta(\log n^n) \rightarrow \Theta(n \log n)$

- Is there such a thing as a sorting algorithm which is **not** comparison-based?

  Radix Sort



LSD Radix Sorting:
Sort by the last digit, then
by the middle and the first one

never directly compare the numbers

- What is the runtime of Radix Sort?

  $\Theta(n \cdot \#)$

  ↑ phases / maximum digits in any one number

- Is this better than MergeSort?

  Depends on digit of the number

- What is the best achieveable runtime to sort an array of size n, which contains the numbers 1 through n randomly permuted?

  linear time (∵ 已知答案是 1, ⋯, n , 只是要挨个 print out)

Graphs

assume no multi-edges , no self-loops

An unweighted graph G is defined by two sets, $G = (V, E)$. V = vertices/nodes. E = edges between pairs of nodes. $n = |V|, m = |E|$.

Some questions:

- What is a **simple graph**? no multi-edges, no self-loops

- What is a **simple cycle**? no repeat nodes

  = cycle <=> circuit (allow repeat nodes)

  { walk : can repeat nodes
  path : can not repeat nodes

- When is an undirected graph a **tree**?

  n-1 edges, connected, no cycle (any 2 imply the third)

- What basic search algorithms can you use to determine if there is a path between two nodes s, t?

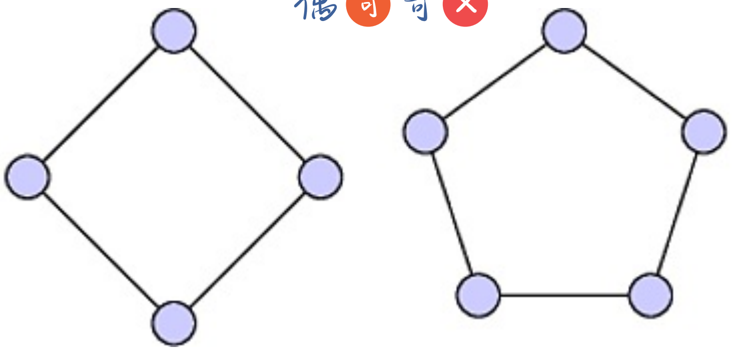
  { BFS → find shortest path
  DFS

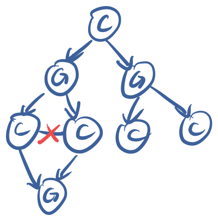
  runtime : $\Theta(m+n)$

- What are the running times of these search algorithms? $\theta(m+n)$
- Which of these algorithms work if you are specifically looking for the shortest path? BFS

A graph is **bipartite** if nodes can be colored red or blue such that all edges have one red end and one blue end. no mono-colored edges

偶 可 奇 ✗





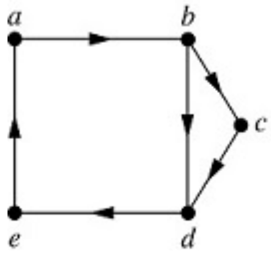
Bipartite Graph Questions:

- What was wrong with the second example? Why was it impossible to color correctly? odd length cycle
  If a graph has an odd-length cycle, then is not bipartite ✓
- Is the converse of that statement necessarily true?
  converse: If a graph is not bipartite, then it has an odd length cycle ✓
- How could one write an algorithm to test whether a graph is bipartite? This should be based on a basic search algorithm.
  use BFS fail: find an edge between two nodes in the same layer of a BFS tree

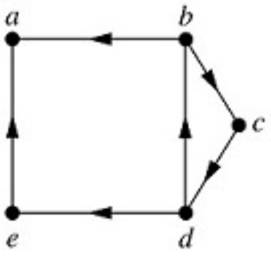
A directed graph is **strongly connected** if every pair of nodes can reach each other. 一定形成2k+1 length 的 cycle

A directed graph is **connected** if for every pair of nodes, one can reach the other.

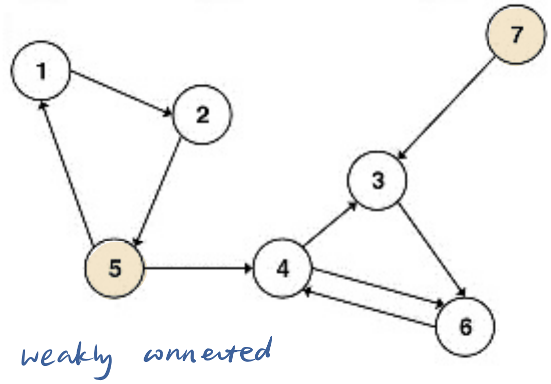
A directed graph is **weakly connected** if every pair of nodes can reach each other if you ignore edge directions.



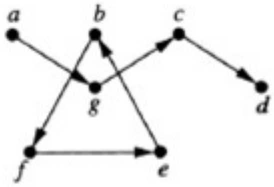
strongly connected



connected



weakly connected



disconnected

Connectivity Questions:

- True Is the following statement true or false? Let s be an arbitrary node. G is strongly connected iff every node is reachable from s and s is reachable from all nodes.
  $\rightarrow$ ✓ $\leftarrow$ ✓ everything can just go through s
  v $\uparrow\downarrow$ $u \rightleftarrows s$
- How could one write an algorithm to test whether a graph is strongly connected?
- Can we improve the efficiency of our algorithm?

Extra Problems:

- Chapter 3, exercises 2, 4, 5, 7, 9, 10
- Challenge problems: Chapter 3, exercises 6, 12

Algorithm :

choose s
run BFS (DFS) to verify everything is reached
For each node $v \neq s$:
  run BFS(DFS) from v, check that s is reached

$\Theta(n(m+\underline{n}))$
$\approx \Theta(nm)$ → mark all nodes as unvisited

But if $m << n$ → highly disconnected
→ fails very soon
→ don't have to include $\Theta(n)$ n times
(∴ $+n$ 可有)

⇒

improve :

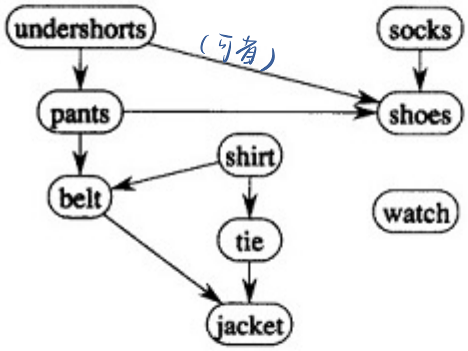
reverse all edges directions
$\Rightarrow G^R$
run BFS from s on G^R

s 如果能 get v on G^R
那么在 G 上 v 能 reach s

$\Theta(m+\underline{n})$
↳ set all points to unvisited

A **directed acyclic graph** (or DAG) is a directed graph with no directed cycles.

A **topological order** of a directed graph is an ordering of its nodes $v_1, ..., v_n$ such that for every edge (v_i, v_j), $i < j$.



Questions:

- How are these two concepts related?

  if a graph has a topological order, then it's a DAG.

- Does every DAG have a topological ordering?

  Yes

no failing condition
↓
prove

- True True or False: If G is a DAG, then it has a node with no incoming edges.

  proof by contradiction:

  assume every node has some incoming edges.

  pick one node, go through it's incoming edges.

  回起点 / infinite node (x) ⇒ 回起点

- Design an algorithm to find a topological ordering, given the above idea.

  start with the node having no incoming edges node 1 → $\Theta(m+n)$

  remove

  then update incoming edges, find next node w/o incoming edges node 2

  ...

  worst case: n 次

use induction

- What is the runtime of the algorithm?

  $\Theta(n(m+n))$

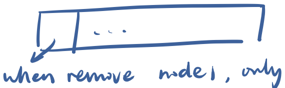
- What would this algorithm look like if we calculated the topological order in reverse?

  => $\Theta(m+n)$ in total

- What basic algorithm would aid in finding the topological order in reverse?

improve:

set queue of nodes w/o incoming edges



when remove node 1, only check directed nodes from node 1,
decrement their # of incoming edges
if the node has 0 incoming edges, add them to queue

```
walk through graph, calculate in-degree of all nodes    m+n
    keeping track of which nodes have no incoming edges    n
    add them to queue
while ( thing in queue ) :    n times
    dequeue
    decrement # of incoming edges for nodes pointed by the prev node
    if then # of incoming edges == 0    $d^+$(node next)
    add to queue.
                                        => $\sum_v d^+(v) = m$
```

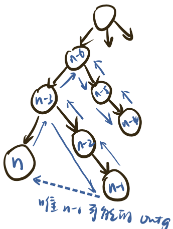
In reverse:

找 node with no outgoing edge (倒着找, 从最后一个点开始找)

找 dead-end, 然后 backtracing, 如果上一个 node 有其他路, 就往下走, 直到 dead-end

(可无视 visited 过的点)

use DFS



唯 n-1 可能的 outgoing edge

$\Theta(m+n)$

Dynamic Programming

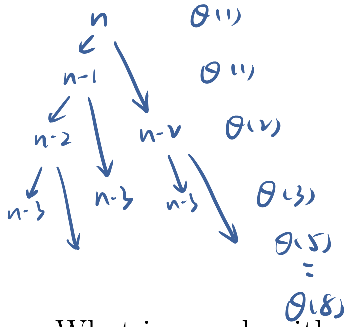
```
**Fibonacci**(int n)
If $n \le 2$ Then Return 1
Return **Fibonacci**(n-1)+**Fibonacci**(n-2)
```

Calculating Fibonacci Numbers

- What is the recurrence relation for this algorithm?

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

- Can Master Theorem solve this recurrence? No



runtime = sum of first n Fibonacci numbers

*所以 recursion 最好用于与除法相关的计算

- What is my algorithm doing which is rather stupid?

  重复计算

- How could I fix this problem?

  利用 recursive <=> iterative

```
**DynamicProgrammingFibonacci**(int n)
$A[1] = 1$
$A[2] = 1$
For $i = 3$ to $n$
    $A[i] = A[i-1] + A[i-2]$
Return $A[n]$
```

$\Theta(n)$

(其实只用存 2 int)

Memoization is the process of writing down intermediate results to refer to later.

Dynamic Programming is the process of transforming a recursive function which replicates work into an iterative one which solves each subproblem once, writing down the answer for future reference.

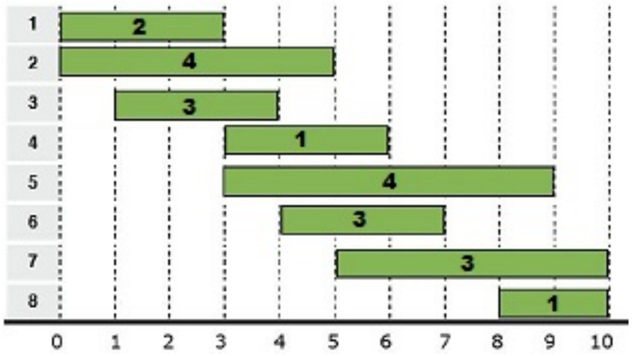
"Write Stuff Down"

先写 recursive 再转动规

Dynamic Programming → Limited Brute Force Search

In the **Weighted Interval Scheduling** problem, we have n jobs. Job j has start time s_j, finish time f_j and value v_j. Two jobs are incompatible if their times overlap. Find the max-valued subset of mutually compatible jobs.

Byte-Size Decision



What is the optimal solution for the above instance? 2, 7 / 3, 6, 8

Finding the optimal solution in general is a daunting task. It is always a good idea to split the problem up into smaller, bite-size pieces.

Question 1: Do I include interval 1 in my solution or not?

Yes → Q4
No → Q2

a = (solve for Q4) + v1
b = (solve for Q2)
return max(a, b)

Question 2: Do I include interval 2 in my solution or not?, etc.

The top level of our recursion will tackle the first piece. Each successive function call will tackle a later piece. We *do not yet know* if we should include interval i. Therefore, we will try including it, and try not including it, and determine which of the two produce a better solution overall.

- If I **do not** include interval 1, what is the *next* question which should be asked?
- If I **do** include interval 1, what is the *next* question which should be asked?
- If x is the best value that can be obtained on intervals 4 through 8, then what is the best value that can be obtained if I include interval 1?

We will define $S[z]$ to be the first interval i such that $s_i \geq f_z$. We are assuming that the intervals are sorted by s_i, and that we fill this array out prior to running our recursive algorithm. We will define $S[z] = n+1$ when there are no intervals that satisfy this condition.

```
WIS(integer $z$)
If $z > n$ Then Return 0
$x = $ WIS$(z+1)$
$y = v_z + $ WIS$(S[z])$
Return max$(x, y)$
```

→ first interval whose start j ≥ finish z

use Memorization

先算后面的，后用于前面的计算

↓
反向 iterate

After this point we want to unroll this into a iterative solution which does not repeat work.

```
WIS(integer $z$)
1: $W[n+1] = 0$    用W[]存每个pos起始后的最佳解
2: For $z = n$ to 1
n ( Θ(1) 3:    $W[z] = \max(W[z+1], v_z + W[S[z]])$ 倒着算
4: Return $W[1]$
```

=> Dynamic Programming Part : Θ(n)

1. This is our base case from the recursive function. W[n+1] = 0

2. Here we are identifying the order in which we fill the array. We need to identify the order such that we have the information we need when we need it. If we tried to fill this array in from 1 to n, we would have failed.

3. We're calculating $W[z]$ in the **exact** same way we calculated WIS(z), except we have the values $W[z+1]$ and $W[S[z]]$ written down (memoized) for future use, instead of recursively calculating them on the spot.

4. After we have found our array, we need to identify where the answer is stored in the array, and return that value.

← ← ←

7	7	7	4	4	4	3	1	0

You can write your solutions in the above manner if you wish. I'll go through the exact same process, but in a less formal, more "pseudo-code" type description, which you are encouraged to use. This is generally easier to read and grade. You must identify 5 things in such a description:

1. State what parameters your function accepts, and what value it returns. We are not mind-readers, so please explain what you are doing.
   define input, output parameters

2. Give the recursive function to calculate your intended function output. This is usually the hardest step.

3. Give the base case(s) for your recursive function.

4. State what order you will fill the array in for your iterative procedure.

5. State where the answer is stored in your final array.

*6. Runtime

```
Let $\text{WIS}[z]$ contain the best value which is attainable on intervals $z$ through $n$.
$\text{WIS}[z] = \max(\text{WIS}[z+1], v_z + \text{WIS}[S[z]])$
$\text{WIS}[n+1] = 0$
Fill the array in reverse order from $\text{WIS}[n+1]$ to $\text{WIS}[1]$
The answer is stored at $\text{WIS}[1]$.
```

Dynamic Programming Design Process

1. Reduce the problem to a series of ordered decisions (What is the nth, (n-1)st, (n-2)nd, etc, fibonacci number? Do we include interval 1, 2, 3, etc?). The simpler the decisions are, the better.

2. Make sure your subproblems can be described concisely: this description is what we pass down to the next level of recursion (We only need to remember which fibonacci number we're currently calculating. We only need to remember which interval we're currently considering).

3. Design a recursive procedure to make these decisions in an ordered way.

4. Instead of using recursion, show how to solve this problem by iteratively filling out a table.

Why do you suppose this programming technique is called **Dynamic** Programming?

Extra Problems:

- Chapter 6: exercises 1, 4, 6, 19, 20, 26, 27

- Challenge problems: Chapter 6: exercise 24

Lecture 8: Dynamic Programming

You can write your solutions as code if you wish. I'll go through the exact same process, but in a less formal, more "pseudo-code" type description, which you are encouraged to use. This is generally easier to read and grade. You must identify 5 things in such a description:

1. State what parameters your function accepts, and what value it returns. We are not mind-readers, so please explain what you are doing.
   define input, output parameters
2. Give the recursive function to calculate your intended function output. This is usually the hardest step.
3. Give the base case(s) for your recursive function.
4. State what order you will fill the array in for your iterative procedure.
5. State where the answer is stored in your final array.

* 6. Runtime

1: Let WIS$[z]$ contain the best value which is attainable on intervals z through n.

2: $\text{WIS}[z] = \max(\text{WIS}[z+1], v_z + \text{WIS}[S[z]])$

3: $\text{WIS}[n+1] = 0$

4: Fill the array in reverse order from WIS$[n+1]$ to WIS$[1]$

5: The answer is stored at WIS$[1]$.

Dynamic Programming Design Process

1. Reduce the problem to a series of ordered decisions (What is the nth, (n-1)st, (n-2)nd, etc, fibonacci number? Do we include interval 1, 2, 3, etc?). The simpler the decisions are, the better.
   Reduce the problem to byte-size decisions.
2. Make sure your subproblems can be described concisely: this description is what we pass down to the next level of recursion (We only need to remember which fibonacci number we're currently calculating. We only need to remember which interval we're currently considering).
3. Design a recursive procedure to make these decisions in an ordered way.
4. Instead of using recursion, show how to solve this problem by iteratively filling out a table.

Longest Increasing Subsequence

Given a sequence of numbers $s_1, s_2, ..., s_n$, delete the fewest numbers possible so that what is left is in increasing order.

Example input: $3, 4, 1, 2, 8, 6, 7, 5, 9$

We'll try the same bite-size questions we have in the previous problems.
Q1: Do we keep s_1 or not? can do LIS (pos, lower limit) ✓ would work

↳ not good runtime

- If I **do not** keep s_1, which number should I try next? s_2
- If I **do** keep s_1, which number should I try next? next[s_1] = s_2

  ↳ just bigger than s_1
- What information must be passed down to the next level of recursion?
- Is this information **concise**? $\text{LIS}[x] = \underbrace{\max(\text{LIS}[x+1]}_{\text{可能会break increasing}}, 1+\text{LIS}[\text{first thing bigger than } x])$ ✗

We lost some critical information: the largest number we've included so far. This information needs to appear in the parameters, or else we will have to pass a ton of information down the recursion chain.

Attempt 2 bite-size questions: ↘
Qi: If I include s_i, what number should I include next?

$k > i,\ s_k > s_i$

把枚举变为DP的关键

☆ 找约束条件！

LIS[z] will store the length of the longest increasing subsequence which starts with s_z.

LIS[$n + 1$] = 0 $\quad \text{LIS}[z] = \max_{k: k>z,\ s_k>s_z} ([1 + \text{LIS}[k]])$

一直是找可叠加的阶段最优解

- I don't know which number to include after s_z, so I try all of them. Well, not really all of them, there are some constraints. What constraints do I place on which number I can include next?
- What does the recursive formula look like for LIS[z]? $\text{LIS}[z] = \max_{k: k>z,\ s_k>s_z} ([1 + \text{LIS}[k]])$
- What order do I fill in the array? 逆序
- Where is the answer stored in the completed array?

  Find the max of the LIS array
- How do I reconstruct the actual sequence?
- What is the runtime of the algorithm?

```
LIS[n+1] = 0
for(z = n to 1):
    LIS[z] = max_{k: k>z, s_k>s_z} (1+ LIS[k])
return max of LIS[]
```

$\theta(n^2)$

$3, 4, 1, 2, 8, 6, 7, 5, 9$
5 4 5 4 2 3 2 2 1 0

can save pointers to get the values.

```
Primality(int X)
For $i = 2$ to $X - 1$
    If $X\%i = 0$ Then Return False
Return True
```

→ length of input is actually $\log_2 x$

电脑是 binary的

- What is the runtime of the above algorithm?

  $\Theta(x)$ $x = 2^{\log_2 x}$ ⇒ runtime is exponential / pseudo-polynomial ↑ polynomial to value

- Is this a polynomial-running time algorithm? 这里是关于 value, 不是 # of input ⇒ no

  No. ↳ has to be linear in relation to the input

- When I ask if an algorithm is polynomial, it must be in relation to something. In relation to what, specifically?

  # of input

Coin-Changing:

There are n denominations of coins $1 = d_1 < d_2 < ... < d_n$ (you have an unlimited amount of each coin). You have a target sum T. Determine the fewest number of coins needed to make change.

- Suppose I'm using American currency ($d_1 = 1, d_2 = 5, d_3 = 10, d_4 = 25$). What would be an optimal algorithm?

  先尽量找最大的能够给几个，其次次大的…

- Suppose my currency is $d_1 = 1, d_2 = 10, d_3 = 15$. Does the above algorithm still work?

  No Eg: $T = 20$

- I need to break this up into bite-size decisions. What should my first decision be?

  whether you take this coin

- What information do I need to pass down to the next level of recursion?

$CC[x]$ will be the fewest number of coins needed to make change for x cents.

- If I select denomination d_i next, what recursive call should I make?

  $CC[x] = \min_{i : d_i \leq t}(CC[t - d_i]) + 1$ $CC[0] = 0$

- If I select denomination d_i next, how does this change the number of coins I've used?

  +1

- At what point should I stop recursing?

  went to 0

$CC[x] = 1 + \min_i(CC[x - d_i])$
$CC[0] = 0$
$CC[x] = \infty$, for all $x < 0$.

CC[0]
from (0 to T)
$CC[t] = 1 + \min_{i : d_i \leq t} CC[t - d_i]$

return CC[T]

- What order should I fill the array?

  from 0 to x

- Where is the answer stored?

  CC[x]

- What is the runtime of the algorithm?

  $\Theta(n 2^{\log_2 T})$ $(= \Theta(nT))$

- Is this a polynomial-running time algorithm?

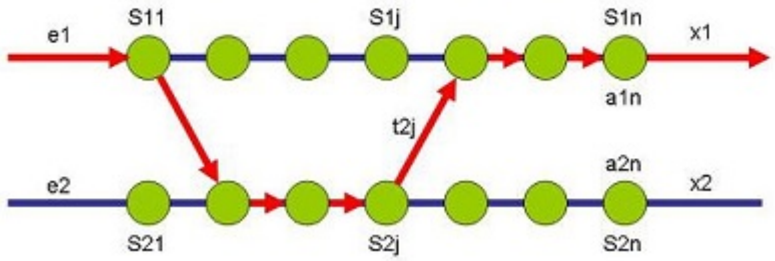
  No.

Lecture 9: 2-D Dynamic Programming

Assembly-Line Scheduling

There are 2 assembly lines, each with n stations. The ith station on each line is denoted $S_{1,i}$ and $S_{2,i}$. An automobile starts at your choice of $S_{1,1}$ or $S_{2,1}$. Station $S_{i,j}$ has a processing time $a_{i,j}$. After station $S_{i,j}$, the automobile can stay on the same line and go to $S_{i,j+1}$ at no cost, or switch lines at cost $t_{i,j}$.

$a_{i,j}$ = processing time



$S[i][j] = \min(S[\text{the other line}][j+1] + t[i][j], S[i][j+1]) + a_{i,j}$

- If we want to find the shortest path across the assembly lines, what basic algorithm could we use to solve this? Dijkstra $\theta(n\log n)$
- We need to break this up into a series of ordered decisions. What should our first decision be? Our second decision?
  switch or stay on current line
- At each level of the recursion, we will be answered a single decision. What information changes at each level of the recursion, and thus must be passed down as parameters?
  ALS(i,j) : i is the line number, j is the station
- If I am at $S_{i,j}$, what cost will I incur regardless of my choice?
- If I am at $S_{i,j}$, and I decide to stay on the same line, what station will I go to next?
- If I am at $S_{i,j}$, what cost will I incur if I decide to switch lines?

Define: ALS$[i, j]$ will store the length of the shortest path from $S_{i,j}$ to the exit.

简化成一个

$$\text{ALS}[1, j] = a_{i,j} + \min(\text{ALS}[1, j+1], t_{1,j} + \text{ALS}[2, j+1])$$

$$\text{ALS}[2, j] = a_{i,j} + \min(\text{ALS}[2, j+1], t_{2,j} + \text{ALS}[1, j+1])$$

$\text{ALS}[i,j] = a_{i,j} + \min(\text{ALS}[i,j+1], t_{i,j} + \text{ALS}[3-i, j+1])$

- What is the base case?
  $\text{ALS}[i,n) = a_{in}$
- What order do I fill in the array?
  backward
- Where is the answer stored in the completed array?
  ALS[1][j] or ALS[2][j]
- What is the runtime of this algorithm?
  $\theta(2ij)$
- How do I determine the actual path through the assembly lines?
  存 choice of each step

```
for j = n to 1
    for i = 1, 2
        算 ALS[1][j], ALS[2][j]
```

Sequence Alignment

We are given two strings $X = x_1 x_2 ... x_n$ and $Y = y_1 y_2 ... y_m$, and we want to determine how similar these strings are.

可 insert/remove

X=ocurrance
Y=occurrence

- If we simply check how many positions i satisfy $x_i = y_i$, what will we determine is the difference between these two strings?

- The above metric isn't very intelligent. What number should we really return?

min # of edits to transform x to y after insert space / remove char

The **edit distance** between two strings is the minimal distance possible between two strings after inserting your choice of spaces. Our goal is to efficiently calculate the edit distance between X and Y.

X=oc-urrance
Y=occurrence

min(insert , remove)

Edit distance = 2

SA[0][i] - edit distance after insert space after i
SA[1][i] - edit distance after remove ith char

- We need to break the problem into bite-size decisions. What should be our first question?
  insert / remove / replace / match
- The top level of recursion will handle the first decision only. What information do we need to pass down to the next level of recursion?
  pos for x and pos for y

Define: $SA[i, j]$ is the min cost of aligning strings $x_i x_{i+1} ... x_n$ and $y_j y_{j+1} y_m$.

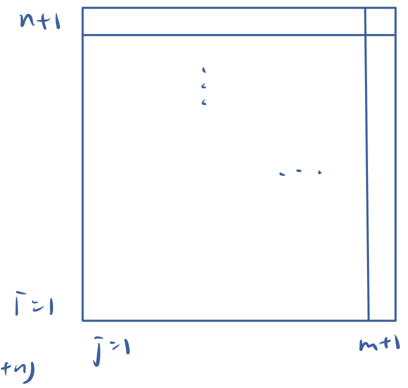
- If $x_i = y_j$, what recursive call should I make?
  SA[i+1, j+1]
- What cost is incurred if $x_i \neq y_j$?
  1
- If I accept the mismatch between x_i and y_j, what recursive call should I make?

- If I insert a gap in the X string, what recursive call should I make?

- Under what conditions should I stop recursing?

if $x_i \neq y_j$, $SA[i,j] = 1 + \min \begin{cases} SA[i+1, j+1] & \text{(replace)} \\ SA[i+1, j] & \text{(remove)} \\ SA[i, j+1] & \text{(add } y_j \text{ to } x\text{)} \end{cases}$

$SA[i,j] = SA[i+1, j+1]$, if $x_i = y_j$
$SA[i,j] = 1 + \min(SA[i+1,j], SA[i,j+1], SA[i+1,j+1])$, if $x_i \neq y_j$.
$SA[i, m+1] = n - i + 1$
$SA[n+1, j] = m - j + 1$ > inclusive ∴要+1



- What order do I fill the array in?

  for j = m to 1
      for i = n to 1

- Where is the answer stored?

  SA[1][1]

- What is the runtime?

  θ(mn)

- What are the space requirements?

  θ(mn) 不太好 => 算一排丢一排 θ(m+n)

	A	C	A	C	A	C	T	A	
A	2	3	3	4	5	6	7	7	8
G	2	2	3	3	4	5	6	6	7
C	2	1	2	2	3	4	5	5	6
A	3	2	1	2	2	3	4	4	5
C	4	3	2	1	2	2	3	3	4
A	5	4	3	2	1	2	2	2	3
C	6	5	4	3	2	1	1	1	2
A	7	6	5	4	3	2	1	0	1
	8	7	6	5	4	3	2	1	0

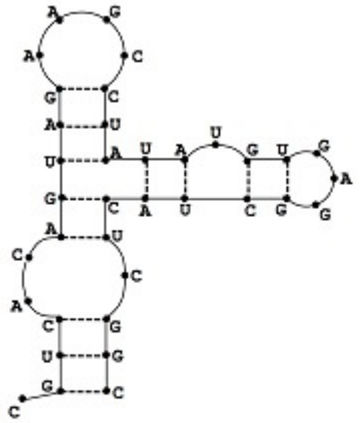
This algorithm is used in computational genetics, which has absolutely huge strings.

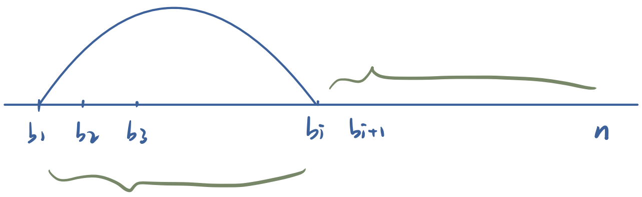
- How could one reduce the space requirements?

- Are there any drawbacks to this solution?

RNA Secondary Structure

RNA is a string $B = b_1 b_2 ... b_n$ over the alphabet $\{A, C, G, U\}$. RNA is single-stranded (as opposed to DNA which is double-stranded), and loops back and forms pairs with itself. The "secondary structure" is determined by figuring out what the pairings are.





link b_1 - b_i will break it into b_2 - b_{i-1} , b_{i+1} - b_n

In reality, how RNA bonds with itself is very complicated. We will simplify it a bit in order to get an approximation of how the structure will look. We will assume the following 3 rules are always followed:

- Each pairing is AU or CG.
- The ends of each pair are separated by at least 4 bases: if $(b_i, b_j) \in S$, then $i < j - 4$.
- No pairs cross each other: if $(b_i, b_j), (b_k, b_l) \in S$, it is not the case that $i < k < j < l$

We will assume that the RNA strand forms the maximum possible number of pairs according to the above rules.

- What should our first decision be?

  Bite size decision #1 : what (if anything) does b_1 pair with ?

- What information do we need to pass down to the next level of recursion?

  endpoints

$RNA[i,j]$ = max number of pairings for string $b_i b_{i+1} \ldots b_j$. 区间 DP

max

- If we do not pair b_i with anything, what recursive call should I make?

  RNA (i+1, j)

- If we pair b_i with something, how does the number of pairs found change?

  $RNA(i,j) = 1 + \max_{k: i+4<k\le j,\ (b_i,b_k) = AU \text{ or } CG} (RNA(i+1, k-1) + RNA(k+1, j))$

- What subproblems (plural!) do we need to consider if we pair b_i with b_j?

  separated by at least 4 bases

  AU / CG

- Under what conditions should we stop recursing?

$$RNA[i,j] = \max(RNA[\underline{i+1}, j], 1 + \max_t(RNA[\underline{i+1}, t-1] + RNA[\underline{t+1}, j]))$$

Base case: $RNA[i,j] = 0$, if $i \ge j - 4$

∴ larger i to smaller i

⇒ for i=n to 1

- What choices of t are valid?

  $i+4 < t \le j$, $(b_i, b_t) = AU$ or CG

- What order do I fill the array in?

  ```
  for i=n to 1
      for j=i to n
  ```

- Where is the answer stored?

  RNA [1, n]

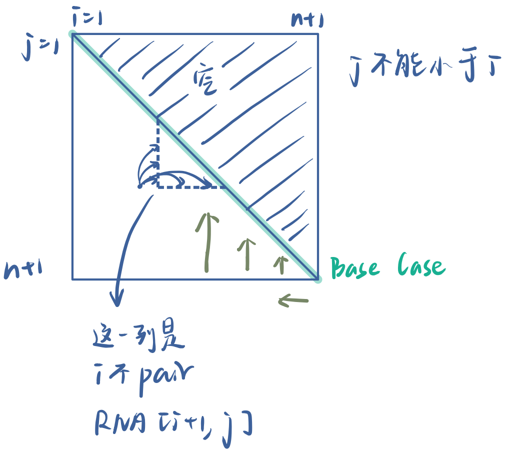
- What is the runtime?

  $\Theta(n^3)$

  ↳ 双层loop (每一个 i,j 遍历全部的 t > i)

  $= \Theta(n^2 \cdot n)$

  $= \Theta(n^3)$



CSCI 270 Lecture 11

Subset Sum

Given positive integers $w_1, w_2, ..., w_n$ and a target W, is there a subset of the integers which add up exactly to W?

Sample instance: integers = $\{2, 5, 7, 13, 16, 17, 23, 39\}, W = 50$

- Is there a subset for the above instance?

  2 { take ss[1, 50-2]
  not ss[1, 50]

- I need to break this up into bite-size decisions. What should my first decision be?

  take / not take

- What information do I need to pass down to the next level of recursion?

  target sum, pos

$SS[x, t]$ will store 1 if you can use a subset of the integers $w_x, w_{x+1}, ..., w_n$ to add up exactly to t. 0 if otherwise

- If I do not include w_x, what recursive call should I make?

  ss [x+1, t]

- If I do include w_x, what recursive call should I make?

  ss [x+1, t-w_x]

- If one answer returns 1, and the other answer returns 0, then what should my recursive function return? 1

- Under what base case conditions do I return 1?

  有1

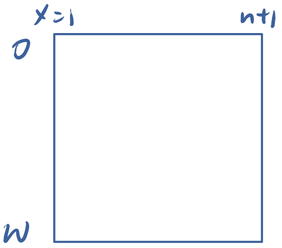
- Under what base case conditions do I return 0?

  无1

$$SS[x,t] = \max(SS[x+1,t], SS[x+1,t-w_i])$$

base case { $SS[x,0] = 1$

$SS[n+1,t] = 0, \forall t \neq 0$



- What order should I fill the array?

  for x= n+1 to 1

      for t=0 to w   calc ss[x,t]
- Where is the answer stored?

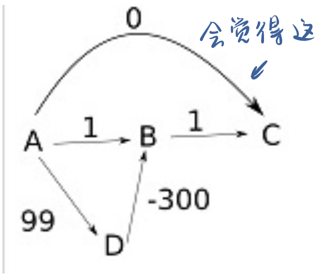
  ss[1, w]
- What is the runtime of the algorithm?

  $\theta(nw)$
- Is this a polynomial-running time algorithm?

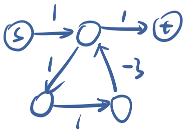
  No (∵ w can be too large)

Shortest Path (Again!)

☆ Allow negative weight



会觉得这是shortest

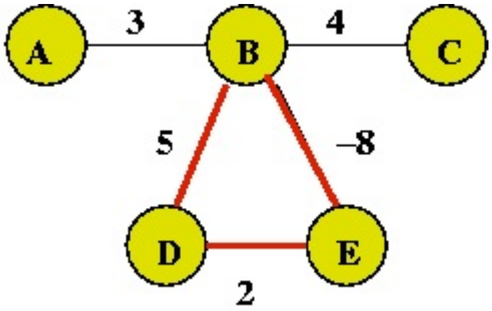


→ no solution

infinite loop on the negative cycle

problems

- What is the length of the shortest path from A to C? A→D→C
- What will Dijkstra's Algorithm find? A→C
- Why didn't Dijkstra's Algorithm work? ∵有 negative edge



What is the length of the shortest path from A to C? 无解

infinite loop on negative cycle

We will assume there are no negative weight cycles, since this leads to nonsense answers and situations.

We want to find the length of the shortest path from s to t on a graph with no negative weight cycles.

- What should my first decision be? which node do I go to next ?
- What information do I need to pass down to the next level of recursion?
  weight

Attempt 1: $SP[x]$ will be the length of the shortest path from x to t.
$SP[x] = \min_{(x,y)\in E}(c_{(x,y)} + SP[y])$
$SP[t] = 0$

when we have loops → infinite recursions

What order should I fill the array?

→ 解决方案: 加 time limit

Attempt 2: $SP[i, x]$ will be the length of the shortest path from x to t using no more than i edges.
$SP[i, x] = \min_{(x,y)\in E}(c_{(x,y)} + SP[i-1, y])$
$SP[i, t] = 0$ 时间内 $t \to t$, weight $= 0$
$SP[0, x] = \infty, \forall x \neq t$

↳ 时间用尽仍未到 t

- What order should I fill in my array?

```
for i = 0 to n-1   ← 上限是 n-1
    for x = s to t
        calc SP[i, x]
```

- Where is the answer stored? $SP[n-1, S]$
- What is the runtime of this algorithm?
  $\Theta(mn)$
- Is this a polynomial runtime?
  Yes ∵ input $= \Theta(n+m)$

This is known as the Bellman-Ford algorithm.

$\Theta(mn)$

BFS : $\Theta(m+n)$

Dijakstra : $\Theta(m \log n)$

CSCI 270 Lecture 12: Greedy Algorithms

All-Pairs Shortest Paths

We want to find the shortest path between all pairs of points (we'll return $n(n-1)$ different answers, one for each pair). BF其实 get all sp from all nodes to one destination (都存在SP里)

How could we do this, using existing algorithms? run n times BF, once per destination $\Theta(mn^2)$

We'll instead calculate this more directly, using dynamic programming, in the hopes that we are able to improve the runtime in some way.

Let $ASP[i, x, z]$ = the length of shortest path from x to z using no more than i edges.

$ASP[i, x, z] = \min_{(x,y)\in E}(c_{(x,y)} + ASP[i-1, y, z])$

- What should my base cases be? $ASP[0, x, y] = \infty$ (if $x \neq y$) $ASP[i, x, x] = 0$
- What order should I fill in the array?
- Where are the answer(s) stored? $ASP[n-1]$ → 出来的一整个 2D-array
- What is the runtime of the algorithm? $\Theta(mn^2)$
- Did we net any improvement in the running time? No

Flond-Warshall

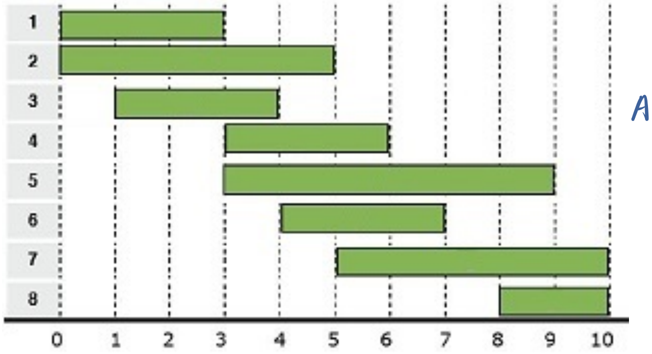
Unweighted Interval Scheduling

Brute Force Search → DP → Greedy

(Unlimited Brute Force Search)

(some criteria, just follow)

We are given n tasks, each with a start time s_i, and finish time f_i. Find the largest subset of tasks such that none of them overlap.



Always start with greedy criteria

- What is the size of the largest possible subset? 3, 6, 8 / 1, 4, 8 / 1, 6, 8
- Propose a greedy criteria. How should we decide which interval to include next? shortest duration, earliest start time, fewest collisions, earliest finish time, latest start time
- Can you find counter-examples to any of these criteria?

无 tie-breaker

一样的，只是方向相反

给 interval 一个 mirror

- Once you've narrowed down your list of candidate algorithms to one, what should you do?

Greedy algorithms are easy to design, easy to write, and very efficient. The tradeoff is that they also tend to be unclear. **Why** does the algorithm work? How can you convince someone that the algorithm is correct?

Greedy algorithms often require a proof of correctness. It is not clear at all that our proposed algorithm for Interval Scheduling is actually correct.

Proving the correctness of an entire algorithm is rather overwhelming, so it is useful to break it down. Just as Greedy Algorithms tackle each bite-size decision sequentially, we will prove the correctness of each of these decisions sequentially, via an inductive proof.

I.H **Inductive Hypothesis:** There is an optimal solution which includes the first k intervals from our solution.

Our first k choices are "right" if there is an optimal solution consistent with those first k solutions

B.C The base case is almost always trivial. Specifically, our base case is $k = 0$.

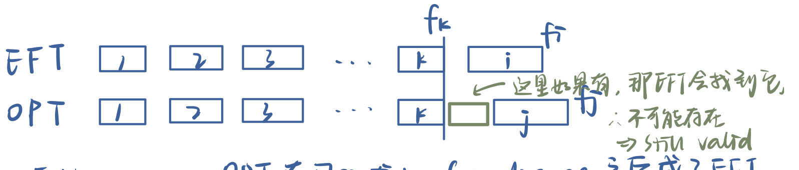
Now we need to show there is an optimal solution which includes the first $k + 1$ intervals from our solution.

What we know:

- There is some interval i which is the $k + 1$st choice in our solution.
- There is at least one (possibly multiple) optimal solutions which include the first k choices we did.
- We need to show at least one of those optimal solutions also include i.

I.S. Assume our $(k+1)^{st}$ choice is not in OPT（假设一个最优解，现阶段存在是抽象的）

Take an arbitrary optimal solution OPT which includes the first k intervals from our solution. Presumably OPT does not include interval i (otherwise we're done). However, of all the intervals it does include (other than the first k), there must be one with smallest finish time. We'll call this interval j.



OPT在iteration of exchange之后成了EFT

每一步如果有不同就替换，最后就会得到EFT的解

① 证 valid

② 证 optimal

- Who do f_i and f_j compare?

  $f_i \le f_j$
- How should I transform OPT?

  $OPT - j + i = OPT'$ ⇒ we have proved our I.H.
- What is the runtime of our algorithm?

  $\Theta(n^2)$

Extra Problems:

☆思想：Bite-size + induction

- Chapter 4: exercises 3, 5, 6, 7, 9, 13, 15, 24
- Prove the correctness of the algorithms you found, using exchange arguments.

Sequential → ☆Exchange Argument: 常用模板

B.C.
I.H.
I.S. OPT - k + i
→ prove valid (not break rule)
→ prove optimal

OPT: what we have get so far, we haven't srewed it up by far.
假设的是一个完整的OPT, 只是在I.S.中我们比的只是含k部分的OPT

CSCI 270 Lecture 13: Sequential Exchange Arguments

- There is an optimal solution which includes the first 0 intervals from our solution (trivial base case).
- Assume there is an optimal solution which includes the first k intervals from our solution, and call it OPT.

☆exchange argument

- Exchange some element of OPT with your algorithm's $(k+1)$st choice, to produce a new solution OPT'. Figuring out which element to exchange is a large part of the challenge.
- Prove that OPT' is still valid; that is, our exchange did not somehow break the rules.
- Prove that OPT' is still optimal; that is, the value of its solution is just as good as OPT.
- You have found an optimal solution OPT' which includes the first $(k+1)$ intervals from our solution!
- Therefore there is an optimal solution identical to ours! Proved by Induction.

Minimum Spanning Trees (MST常见证法: 另一种方法 + 这种方法的一边 → create cycle)

(undirected)

We want to prove that Kruskal's Algorithm is correct, which adds edges from smallest to largest unless adding an edge creates a cycle. We will call the solution produced by Kruskal's Algorithm 'US'.

Base Case: There is an optimal solution which uses the first 0 edges added by US.

Inductive Hypothesis: There is an optimal solution OPT which uses the first k edges added by US.

→ 一个完整的MST, 其中US前k条边在其中

I.S. Suppose US adds edge e_{k+1} next, but OPT does not include this edge.

- Suppose we add edge e_{k+1} to OPT. What happens?

  create a cycle

- There is some edge e_f in the cycle C which is not in US (otherwise we could not have added e_{k+1}). ⇒ e_f must cost more than e_{k+1}, otherwise we would have chosen e_f
- Exchange e_f with e_{k+1} in OPT to create OPT'. $OPT - e_f + e_{k+1} \rightarrow OPT'$
- Prove that OPT' is valid, that is, it is a spanning tree. 3选2: { n-1 edges ✓ / no cycle / connected ✓
- Prove that OPT' is optimal, that is $c(e_f) \geq c(e_{k+1})$.

  → minimum

→ can just use the other part of cycle

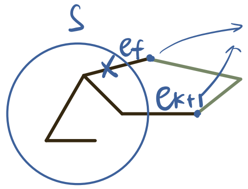
- Assume (by way of contradiction) that $c(e_{k+1}) > c(e_f)$.
- Consider the subgraph of OPT consisting of all edges with cost $\leq c(e_{k+1})$, called OPT_C
- $\{e_1, e_2, ..., e_k\} \cup e_f \subseteq OPT_C$.
- $\{e_1, e_2, ..., e_k\} \cup e_f$ does not contain any cycles, so US would have added e_f before e_{k+1}.
- Contradiction: $c(e_f) \geq c(e_{k+1})$. Therefore OPT' is optimal.

We want to prove that Prim's Algorithm is correct. We will call the solution produced by Prim's Algorithm 'US'.

Base Case: There is an optimal solution which uses the first 0 edges added by US.

Inductive Hypothesis: There is an optimal solution OPT which uses the first k edges added by US.

T.S. Suppose US adds edge e_{k+1} next, but OPT does not include this edge.



只能比这俩

∵ prim's 选 ek+1

不选 ef

说明 ef ≥ ek+1

- Let S be the set of nodes US has connected with the first k edges.
- Add e_{k+1} to OPT to produce a cycle C.
- Since e_{k+1} has one endpoint in S and the other endpoint in $V - S$, part of C is in S and the rest is in $V - S$.
- Therefore there is another edge in C, e_f, with one endpoint in S and the other in $V - S$.
- $e_f \notin \{e_1, e_2, ..., e_k\}$, otherwise S would be a different set of nodes.
- Exchange e_f with e_{k+1} in OPT to create OPT'. $OPT + e_{k+1} - e_f \rightarrow OPT'$
- Prove that OPT' is valid.

  3选2：

  n-1 edges (+1边 -1边)

  no cycle

  connected (can use the rest part of the cycle)
- Prove that OPT' is optimal.

  $e_f \geq e_{k+1}$

  otherwise, prim's algorithm would have chosen e_{k+1}

  ↳ choose the smallest edge to span the tree.

Proof Practice

There are n students and m CPs in CSCI 270. You are forming study groups of size 3, where each group either has 2 students and 1 CP, or 2 CPs and 1 student. Find a greedy algorithm that forms the maximum possible number of study groups.

The Algorithm:
If there are more students remaining than CPs, the next group formed should be of 2 students and 1 CP. Otherwise, form a group of 2 CPs and 1 student.

The Rules:

- Form groups of 2-3 people.
- Prove the correctness of the above algorithm. Your group should write a single solution on a piece of paper. You do not need to write your names. Feel free to ask questions of the instructor.
- When time is up, you will pass your paper to the group specified by the instructor.
- The instructor will now go over a correct proof.
- Read the proof you have been given. Write any comments you like. Put a grade on the top of the paper between 0 and 10 (10 is a perfect proof). Try to consider beyond whether the proof matches the instructor's proof, and instead consider whether the given proof is a valid and rigorous proof.
- Continue to pass each paper you grade to the same group, and continue to grade each paper you receive, writing your grade and comments next to the ones already given by the previous groups.
- If you spend less than a couple minutes grading a paper, you're doing it wrong: take as much time as you need. If you are currently grading one paper, and have at least 2 papers waiting in your queue, dequeue the extra papers and pass them to the next group.
- If you receive your own paper, you can leave after grading any other papers you've already received.
- The purpose of this exercise is to read and grade proofs, and to gain a better appreciation of what constitutes an effective proof. The grades you receive on your paper are of little consequence (you may find that you completely disagree with some of the comments and grades you receive: this is to be expected).

B.C. The solution is same to OPT for 0 groups

(no group to mess up)

I.H. There is an optimal solution OPT which contains the first k groups of this algorithm

ALG[k] = OPT[k] for all 0 to k

I.S. consider the $k+1^{th}$ group

Assume ALG[k+1] ≠ OPT[k+1]

Suppose our choice: 2 stu + 1 cp (∵ now # stu > # cp)

OPT: 1 stu + 2cp → remaining at least 1 student

swap: OPT' = OPT - 1cp + 1stu

validity: 剩余人数不变

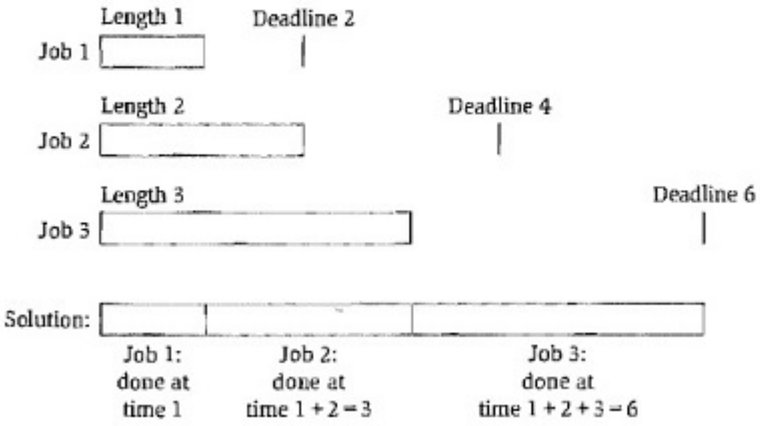
optimal: fewer students remaining

→ more optimal

CSCI 270 Lecture 15: Scheduling to Minimize Lateness

We have n tasks, task j has duration t_j and deadline d_j. We choose an order to execute tasks, and we cannot interrupt a process until it is finished. As soon as one task finishes, we start the next one. If a task finishes after its deadline, it is late. We are concerned about which task is the **most** late. We want to minimize how late this task is.

If we start task i at s_i, then task i will finish at $f_i = s_i + t_i$. The lateness is $L_i = \max(0, f_i - d_i)$. $\max_i L_i$ indicates how tardy the **most** late task is, which we want to minimize.



- What greedy criteria might work for this problem? earliest deadline, shortest duration smallest-slack first ($d_i - t_i$)
- Can you find any counter-examples to these algorithms?

for shortest-slack: (×)

	d_i	t_i
j1	2	1
j2	3	2
j3	3	2

①→②→③

for shortest-duration: (×)

	d_i	t_i
j1	100	1
j2	5	5

EDF 2→1 ✓
SDF 1→2 ×

Proof:

∵这里要的是order，所以可能出现顺序不同，不存在include与否的问题

For Ordering Problem !!!

For a given schedule S, an **inversion** is a pair of jobs i and j where $d_i < d_j$, but $f_j < f_i$. That is, our greedy algorithm says that we should schedule i first, but S scheduled j first.

Our algorithm is the unique schedule with no inversions (assuming no ties).

We want to prove that there is an optimal solution which has no inversions.

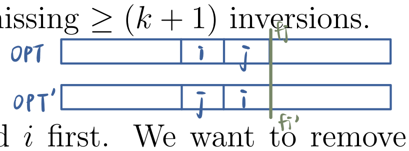
Base Case: There are $C(n, 2) = \frac{n \cdot (n-1)}{2}$ pairs of jobs, and thus $C(n, 2)$ possible inversions. Therefore there is an optimal solution with $\leq C(n, 2)$ inversions.

Inductive Hypothesis: Assume there is an optimal solution with $\leq k$ inversions. (Call it OPT).

Now we need to prove there is an optimal solution OPT' with $\leq (k-1)$ inversions.

Note that if you're uncomfortable doing induction "backwards", you could just as well argue how many inversions are "missing". Your base case would be that there is an optimal solution missing at least 0 inversions, you'd assume there is an optimal solution missing $\geq k$ inversions, and then argue there is an optimal solution missing $\geq (k+1)$ inversions.

Case 1: there is a consecutive inversion $(i, i+1)$.



$fi' = fj$

we did j first

$\Rightarrow dj \leq di$

$\Rightarrow Li' \leq Lj$

(lateness)

i' and j finish at the same time, but i has later ddl

$\Rightarrow$ i' has smaller lateness

Our algorithm schedules $i+1$ first, but OPT scheduled i first. We want to remove this inversion, so swap their positions to produce OPT'. This will have one less inversion, which will contradict our assumption if we can show that OPT' is still a valid optimal solution.

Prove that OPT' is still valid (this one is easy), and prove that OPT' is still optimal (this one is hard).

max late 不变/变小

$\Rightarrow Li \leq Li'$

$Li' \leq Lj \leq v(OPT)$

Therefore, this inversion (i, j) once removed, maintains optimality

- Are the latenesses of any task $< i$ changed from OPT to OPT'?

  No

- Are the latenesses of any task $> i+1$ changed from OPT to OPT'?

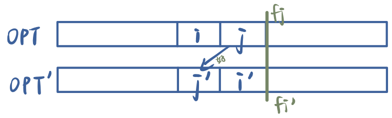
  No

- How does the lateness of task $i+1$ change from OPT to OPT'?

  smaller

- How does the lateness of task $i+1$ in OPT compare to the lateness of task i in OPT'?

- Why is OPT' optimal?



$fj = fi'$

$\because$ j' 先于 i'

$\therefore dj \leq di$

$Li' \leq Lj \leq ans(OPT)$

lateness of i' will be smaller than j

$\Downarrow$

$ans(OPT') \leq ans(OPT)$

$\because j \to j' \therefore Lj' < Lj$

OPT' is still optimal

[类似bubblesort] ③

Case 2: there are inversions, but none of them are consecutive.

Let $(i, i+k)$ be the smallest inversion, where $k > 1$. There is a contradiction in this information: where is it?

case 2 is impossible

5	:	3

无论中间是谁，都可被理解为由更小的 consecutive inversion 组成

5 4 3 → 5 3 4 → 3 5 4 → 3 4 5

3 4 5

---

5 6 3 → 5 3 6 → 3 5 6

3 5 6

# of inversion between OPT and our solution:

$\binom{n}{2} \to \cdots \to k \to k-1 \to \cdots \to 0$ ↗ our solution

最多有 $\binom{n}{2}$ T.H. T.S. (总能到 0)

CSCI 270 Lecture 16: Optimal Caching

A cache can store n items. There is a sequence of m requests $d_1, d_2, ..., d_m$ known in advance. If an item is requested which is not in the cache, it must be brought into the cache, resulting in a **cache miss**. You may only bring something into the cache when it is requested. The goal is to minimize the number of cache misses.

↙ size of cache

$k = 2$, initial contents = 'ab'

requests: a b c b c a a b

online algorithm

In reality : caching problem : don't know requests

a	✓	ab
b	✓	ab
c	×	cb
b	✓	cb
c	✓	cb
a	×	ab
a	✓	ab
b	✓	ab

- What greedy criteria might work for this problem?

  Least frequent in future, Furthest in future (FIF)

  × ✓

- Can you find any counter-examples to these algorithms?

- How is this problem different than the standard version of Caching?

  那个不知道 future

Base Case: There is an optimal solution that has the same cache contents as we do through the first 0 requests.

Inductive Hypothesis: There is an optimal solution OPT that has the same cache contents as we do through the first k requests.

Request $k+1$ is the first request at which the cache contents of US and OPT differ.

- What must happen at request $k+1$? There must be a cache miss at time k+1

(FIF) US

...				
Request k	o_1	o_2	o_3	...
Request $k+1$	d_{k+1}	o_2	o_3	...
...				

d_k : d_f d_o d_e

d_{k+1} : d_{k+1} d_o d_e

Problem :

it can be different in later times for OPT and OPT'

(due to different cache content)

OPT

...				
Request k	o_1	o_2	o_3	...
Request $k+1$	o_1	d_{k+1}	o_3	...
...				

d_k : d_f d_o d_e

d_{k+1} : d_f d_{k+1} d_e → make an exchange to turn it into OPT'

(= FIF)

At request $k+1$, US kicks out o_1, while OPT kicks out o_2. Otherwise, the cache contents are completely identical.

Let's look at what OPT does in the future and see when this change in cache contents becomes relevant. There are many things which can happen for which the change in cache contents has no bearing. If there is a request on o_3, for example, it doesn't matter which of o_1 or o_2 we kicked out. If OPT removes o_3 for d_j, it doesn't matter which of o_1 or o_2 we kicked out.

- What events can occur which makes the different cache contents relevant?

  request for df, request for do, OPT may remove df   其他情况OPT和FIF是一样的

  没想到

- Which of these events could conceivably happen **first**?

  request for do or OPT removes df ⇒ 分类讨论 ↘ only care about the first event

We will refer to the first of these events as having occured at request j.

Case 1: OPT removes $\overset{(df)}{o_1}$. at request j

除了 k+1

假设其他每一步变化OPT都和OPT'一样

⇒ 为了证明 FIF 的k+1那一步可以与OPT一样 optimal

	...	df	do	de	
OPT	Request k	o_1	o_2	o_3	...
	Request $k+1$	o_1	d_{k+1}	o_3	...
	...				
	Request $j-1$	o_1	d_{k+1}	o_3	...
	Request j	d_j	d_{k+1}	o_3	...
	...				

We want to make an exchange, thereby transforming OPT into OPT'. We want OPT' to have the same number of cache misses as OPT. We also want OPT' to be identical to US through $k+1$ requests.

	...	df	do	de	
OPT'	Request k	o_1	o_2	o_3	...
	Request $k+1$	d_{k+1}	o_2	o_3	...
	...				
	Request $j-1$	d_{k+1}	o_2	o_3	...
	Request j	d_{k+1}	d_j	o_3	...
	...				

OPT' can do whatever it wants to fix itself

we can kick out do ⇒ 之后的每一步都一样

⇒ 所以如果 k+1 按FIF的选，FIF的解也存在与OPT一样答案的解

What should OPT' do at request j?

Case 2: There is a request on o_2

	…	df	do	de	
	Request k	o_1	o_2	o_3	…
	Request $k+1$	o_1	d_{k+1}	o_3	…
OPT	…				
	Request $j-1$	o_1	d_{k+1}	~~o_3~~	…
	Request j	o_1	d_{k+1}	o_2	…
	…				

OPT has to kick something out for o_2, we will say it kicks out the here-to-fore unmentioned o_3.

We want to make an exchange, thereby transforming OPT into OPT'. We want OPT' to have the same number of cache misses as OPT. We also want OPT' to be identical to US through $k+1$ requests.

	…	df	do	de	
	Request k	o_1	o_2	o_3	…
	Request $k+1$	d_{k+1}	o_2	o_3	…
OPT'	…				
	Request $j-1$	d_{k+1}	o_2	o_3	…
	Request j	d_{k+1}	O_2	O_3	…
	…				

∵ no cache miss here (one step ahead)

- What would we like OPT' to do this at this request?

  kick out de for df

- Does this break any rules?

  不能，因为无 cache miss

We will delay the exchange until an event occurs. OPT' has o_3, but OPT has o_1. The cache contents are otherwise identical. The possible events are:

- Something (o_4) is removed for o_3. We now plan to kick out o_4 at the next event, instead of o_3. We avoid a cache miss, meaning that OPT wasn't even optimal.

- o_1 is removed for something. We kick out o_3 instead. We avoid a cache miss.

  现在 request df 是可能的 case，因为 request do 已发生

- o_1 is requested. Now we kick out o_3 as originally planned. We simply delayed our cache miss: it occurs for both solutions, but at different times.

- OPT removes O_1 for sth. else

  we can have OPT' remove O_3 for the same thing

?

OPT' doesn't follow greedy criteria

⇒ 我们只证明了 FIF 在 k+1 的解存在 optimal 的可能性

未证 FIF 一定 optimal ?

Divide and Conquer

Divide and Conquer is a **runtime improvement technique**. Generally, we have a basic brute-force/greedy/dynamic-programming algorithm, and then we attempt to improve our performance using Divide and Conquer.

Both MergeSort and QuickSort are examples of Divide and Conquer algorithms. You take the input, split it into pieces, recursively solve each piece, and then figure out how to combine the pieces together. Typically the combine phase is the difficult part.

Divide and Conquer recursion is **completely different** than Dynamic Programming recursion.

- DP is sequential (make the first decision, then recursively make the next decisions).
- Divide and Conquer recursion is parallelized: we split it up into several independent pieces which can be solved in parallel.
- DP recursion can be unrolled into a more efficient iterative algorithm.
- Divide and Conquer recursion cannot be unrolled in a straight-forward manner (it can often be done, but it is usually more trouble than it's worth).

Divide and Conquer : parallelized recursion
no reason to get rid of recursion

[runtime improving technique
can be applied to other algorithms]

DP : sequential recursion
getting rid of recursion → much faster

尝试顺序：DP → greedy → divide and conquer

CSCI 270 Lecture 17: Divide and Conquer

Counting Inversions

You're writing the software for Pandora270. A user ranks n songs, and you want to find other users with similar tastes in music (and recommend music based off these matchings).

Suppose there are 5 songs, conveniently named 1, 2, 3, 4, 5, which I have ranked in that order. You have ranked them in order 1, 3, 4, 2, 5.

How similar are our tastes? In one sense, we have only ranked two songs in the same slot. However, if you look more closely, you'll see we only disagree on one song: song 2.

An **inversion** is a pair of songs i and j such that one user ranks $i < j$ and the other ranks $i > j$. How many inversions are there in the above example?

<2,3> <2,4>   2 out of 10 = 80% similarity

We want to count the number of inversions between two different rankings. For simplicity, we assume that one of the rankings is the numbers 1 through n in increasing order.

We could just solve this in $\theta(n^2)$ time using a brute-force count. Instead, we will use Divide and Conquer to improve our running time.

Your ranking:

1 5 4 8 10 2 6 9 12 11 3 7
1 5 4 8 10 2 | 6 9 12 11 3 7 (Divide: O(1))
5 inversions | 8 inversions (Recursively Solve! $2f(\frac{n}{2})$)

1. Whats our base case?

   one song : 0 inversion

2. Can we just return $5 + 8$?

   No. There can be inversions span accross the border.

3. How long does our combine phase take?

   $\theta((\frac{n}{2})^2)$

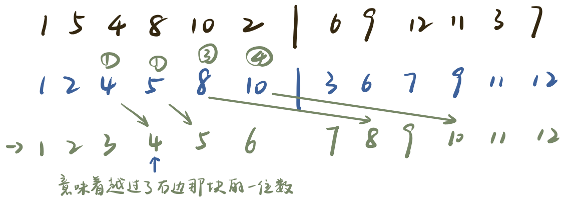
4. What's the resulting recurrence relation?

   $f(n) = 2f(\frac{n}{2}) + \theta(n^2)$

5. What's the overall runtime of our algorithm?

   $\theta(n^2)$

Often your first attempt at a Divide and Conquer algorithm will net no improvement. That doesn't mean Divide and Conquer won't work, it simply means you need to improve a part of the algorithm.



1+1+3+4=9

mergesort - 步是O(n) 双指针

⇒ 5+8+9 = 22 inversions in total

1. How can we improve the combine phase?

   perform the inversions (sort while counting inversions)

   改变的信息是已存的各块 inversion 个数

   ⇒ merge-sort and count-inversions

   合并时数字正位, 向右移时越过几位就加几

2. What's the recurrence relation now?

   $f(n) = 2f(\frac{n}{2}) + \theta(n)$

3. Whats the overall runtime?

   $\theta(n\log n)$

All Pairs Shortest Paths Strikes Back

We want to find the shortest path between all pairs of points (we'll return $n(n-1)$ different answers, one for each pair).

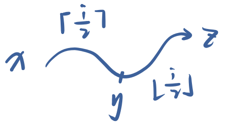
Let $ASP[i, x, z]$ = the length of shortest path from x to z using no more than i edges.

shortest path : $\theta(nm)$

其实找了从 x 出发到所有点的最短路径

$ASP[i, x, x] = 0$
$ASP[0, x, z] = \infty$
$ASP[i, x, z] = \min_{(x,y)\in E}(c_{(x,y)} + ASP[i-1, y, z])$

```
For $i = 0$ to $n - 1$
    For all nodes z
        For all nodes x
            Calculate $ASP[i, x, z]$
```



We can argue the last loop takes $\theta(m)$ time, and there are n iterations for the other loops, so the runtime is $\theta(mn^2)$.

- If we're at node x and we need to get to node z, the dynamic programming way is to find the next node after x that we visit. What would the divide and conquer way be?

  Find the middle node

- Using this idea, what would our new recursive formula be?

  Base case : $ASP[1, x, y] = cost(x, y)$

  only a single edge away

  $ASP[i, x, y] = \min_{y \in V} [ASP[\lceil \frac{i}{2} \rceil, x, y] + ASP[\lfloor \frac{i}{2} \rfloor, y, z]]$

- What values of i do we need to iterate over?

  0 to n-1

- What would our runtime for this algorithm be?

  $\theta(n^4)$

```
for i = 0 to n-1
  for all nodes z        ⇒ θ(n^4) ⍨
    for all nodes x
      θ(n)
```

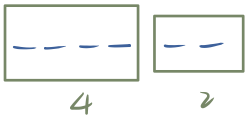
we can just paste 2 parts together

$$ASP[c, x, y] = \min_{y \in V} [ASP[c-1, x, y] + ASP[c-1, y, z]]$$

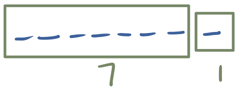
才是真正意义上对半 divide

$ASP[c, x, z]$ = length of shortest path from x to z using $\leq 2^c$ edges

```
for $c = 0$ to $\log n$
    for all nodes $z$          $\Rightarrow \Theta(n^3 \log n)$
        for all nodes $x$
            $\Theta(n)$
```



$= 4 + 4$ 的解



$= 8 + 8$ 的解

Floyd - Warshall

CSCI 270 Lecture 18: The Return of Sequence Alignment

```
       F    L
      [11][00]   x = xF ∘ xL
  ×   [11][01]   y = yF ∘ yL
  ---------------
        1100
       0000
      1100
     1100
  ---------------
   10011100
```

Integer Multiplication

1. Elementary Math time! How do you calculate $12 \cdot 13$? $= 156$
2. How would a computer calculate it? binary
3. What is the running time to multiply two n-bit integers? $\Theta(n^2)$

We're going to try to use Divide and Conquer to improve on this.

Let $x = x_F \cdot 2^{\frac{n}{2}} + x_L$, where x_F is the first $\frac{n}{2}$ bits of x. Similarly, $y = y_F \cdot 2^{\frac{n}{2}} + y_L$.

$xy = x_F y_F \boxed{2^n} + (x_F y_L + x_L y_F)\boxed{2^{\frac{n}{2}}} + x_L y_L$ → 不用计算2^n，只用移位 ⇒ 相加是$\Theta(n)$

n bit × n bit ⇒ 4个 $(\frac{n}{2})$ bit × $(\frac{n}{2})$ bit 的和

$M_1 = x_F y_F$

$M_2 = x_L y_L$

$M_3 = (x_F + x_L) \cdot (y_F + y_L)$

$\Rightarrow x_F y_L + x_L y_F = M_3 - M_1 - M_2$

1. What's our base case? only 1 bit
2. What is the recurrence relation here? $f(n) = 4f(\frac{n}{2}) + \Theta(n) = \Theta(n^2)$
3. What would our runtime be? $\Theta(n^2)$
4. What part of our algorithm needs improvement?
5. How can we improve it? Hint: What is $\underbrace{(x_F + x_L)}_{}(y_F + y_L)$? $(\frac{n}{2}+1)$ bit $\cdot$ $(\frac{n}{2}+1)$ bit $\quad f(\frac{n}{2}+1) \approx f(\frac{n}{2})$

$= x_F y_F + x_L y_L + x_F y_L + x_L y_F$

$\Rightarrow xy = \underbrace{x_F y_F}_{M_1} \cdot 2^n + \underbrace{(x_F y_L + x_L y_F)}_{M_3 - M_1 - M_2} \cdot 2^{\frac{n}{2}} + \underbrace{x_L y_L}_{M_2}$

$f(n) = 3f(\frac{n}{2}) + \Theta(n)$

$\Rightarrow f(n) = \Theta(n^{\log_2 3}) \approx n^{1.57}$

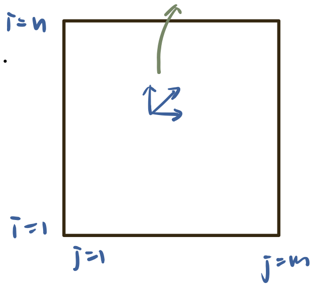
The Return of Sequence Alignment

The **edit distance** between two strings is the minimal distance possible between two strings after inserting your choice of spaces. Our goal is to efficiently calculate the edit distance between X and Y.

Define: $SA[i,j]$ is the min cost of aligning strings $x_i x_{i+1} \ldots x_n$ and $y_j y_{j+1} y_m$.

$SA[i,j] = SA[i+1, j+1]$, if $x_i = y_j$
$SA[i,j] = 1 + \min(SA[i+1,j], SA[i,j+1], SA[i+1,j+1])$, if $x_i \neq y_j$.
$SA[i, m+1] = n - i + 1$
$SA[n+1, j] = m - j + 1$

每一次计算只需要2条相关的，但如果边算边删就不能trace



runtime: $\Theta(mn)$

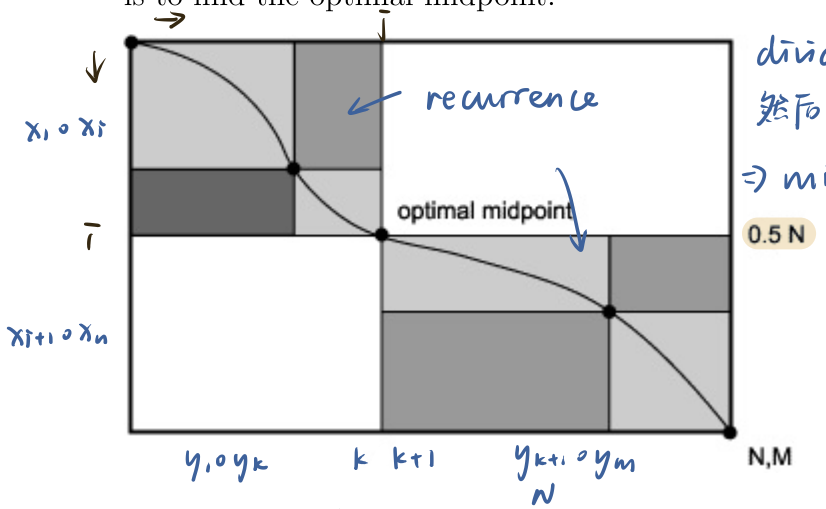
space: ~~$\Theta(mn)$~~ $\Theta(m+n)$

```
For all $i = n + 1$ to 1
    For all $j = m + 1$ to 1
        Calculate $SA[i, j]$
```

Runtime: $\theta(mn)$
Space: $\theta(mn)$

To improve the space requirements, we can toss out old columns when they no longer are relevant. We keep the last two columns only, thereby reducing the space requirement to $\theta(m + n)$ (the size of the input, which you can't improve on). The drawback is you cannot reconstruct the answer, which is a deal-breaker for certain applications.

We will instead use Divide and Conquer to get the best of both worlds. The high level idea is to find the optimal midpoint:



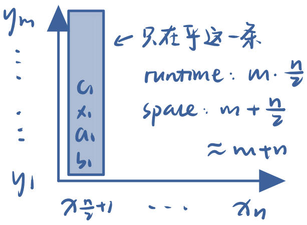
divide x into 2 parts (dp)
然后尝试y中所有分割情况，找最合适的分割线
⇒ minimize the sum of the two parts

Run Sequence Alignment on $\overset{N}{X} = x_{\frac{n}{2}+1}x_{\frac{n}{2}+2}...x_n$ and all of $\overset{M}{Y}$. We save the first column of data (the column which is calculated last), which tells us the optimal matching of the second half of X with any possible suffix of Y.

We want to run Sequence Alignment on $X = x_1...x_{\frac{n}{2}}$ and all of Y to get the other half of the equation. We want to find the optimal matching of the first half of X with any possible prefix of Y.

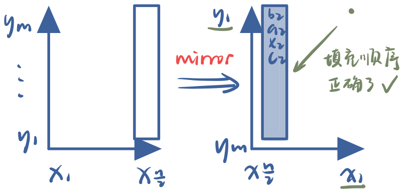
- If we run this normally, you would find the optimal matching of the first half of X with any possible **suffix** of Y.
- We'll reverse Y so that we're matching a prefix of Y, rather than a suffix of Y.
- This twists everything around, however. The first half of X will be matched with the mirror image of Y.
- We have to reverse X too!

① 先算 $y_1 \cdots y_m$ 与 $x_{\frac{n}{2}+1} \cdots x_n$
（因为本来算法就是从后往前）



← 只在乎这一条
runtime: $m \cdot \frac{n}{2}$
space: $m + \frac{n}{2}$
$\approx m+n$

② 算 $x_1 \cdots x_{\frac{n}{2}}$ 与 $y_1 \cdots y_m$ 时用镜像



combine 两条
⇒ minimize sum of 2 parts

runtime: $m \cdot \frac{n}{2}$
space: $m + \frac{n}{2}$ → 只存两条因为要留的信息只有 optimal midpoint
$\approx m+n$

第一次： mn

divide一次后： $\frac{mn}{2}$

divide两次： $\frac{mn}{4}$

⋮

Run Sequence Alignment on $X = x_{\frac{n}{2}} x_{\frac{n}{2}-1} ... x_1$ and $Y = y_m y_{m-1} ... y_1$. Save the first column of data (the column which is calculated last), which tells us the optimal matching of the first half of X with any possible prefix of Y.

→ 就是x对应的y_j ⇒ 也就是我们要的tracing

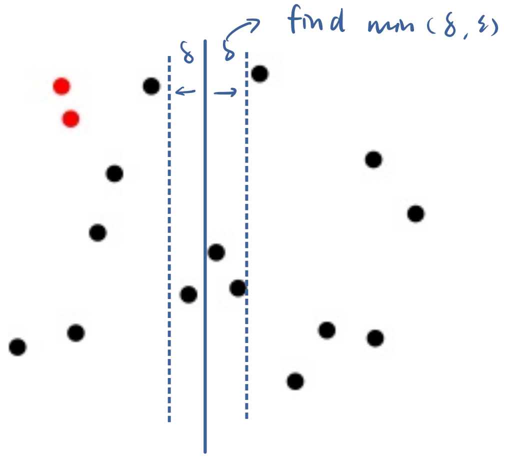
Find the optimal midpoint, and recursively repeat the algorithm!

第二层 runtime： $k \cdot \frac{n}{2} + (m-k) \cdot \frac{n}{2} = m \cdot \frac{n}{2}$; 第三层： $k \cdot \frac{n}{4} + (i-k) \cdot \frac{n}{4} + (j-i) \cdot \frac{n}{4} + (m-j) \cdot \frac{n}{4} = m \cdot \frac{n}{4}$

Closest Points on a Plane

runtime ： $mn + m \cdot \frac{n}{2} + m \cdot \frac{n}{4} + \cdots = 2mn$

Given n points on a plane (specified by their x and y coordinates), find the pair of points with the smallest euclidean distance between them.

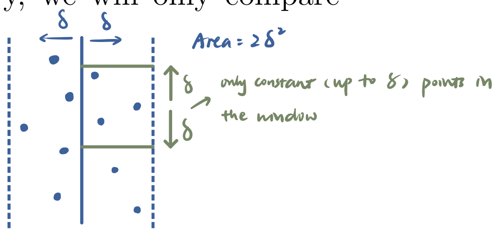


1. What runtime can you achieve simply using brute-force? $\Theta(n^2)$
2. Using Divide and Conquer, how should we divide the plane? make一半在左，一半在右
3. What do we need to do in our combine phase? check distance
4. What's the recurrence relation for our algorithm? $f(n) = 2f(\frac{n}{2}) + \Theta(n^2) = \Theta(n^2)$
5. What runtime does this achieve? $\Theta(n^2)$

Let δ be the min distance between any pair so far. Instead of comparing all pairs of points on each side, we will instead only look at points within δ of the boundary.

What's the worst-case runtime for our new algorithm? $\Theta(n^2)$ 因为可能所有点都在这个区间内

Instead of comparing all pairs of points within δ of the boundary, we will only compare points if their y-coordinates are within δ of each other.



What's the worst-case runtime now?

take all x, sort by x → $\Theta(n\log n)$

take all y, sort by y → $\Theta(n\log n)$

https://aaronice.gitbook.io/lintcode/sweep-line/closest-pair-of-points

CSCI 270 Lecture 19: Network Flow

Network Flow

We have a weighted directed graph $G = (V, E)$, where the edge are "pipes" and their weight is their flow capacity. These values measure the rate in which fluid/data/etc can flow through the pipe. What is the maximum rate that flow can be pushed from s to t in the above graph?

Each edge e has capacity $c(e)$. An **s-t flow** is a function f which satisfies:

max-value flow : max amount of flow from s to t

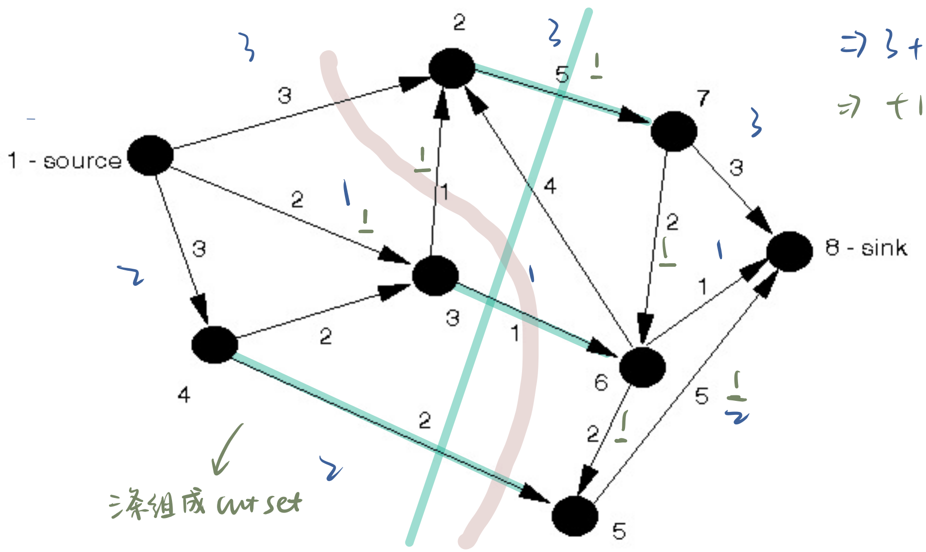
流量限制 1. $0 \leq f(e) \leq c(e)$, for all e (capacity).

↗ 只进不出 / 只出不进

流量守恒 2. $\sum_{(x,v)} f((x, v)) = \sum_{(v,y)} f((v, y))$, for all nodes $v - \{s, t\}$ (conservation).

sum of incoming flow = sum of outgoing flow

The value of a flow is $v(f) = \sum_{(s,x)} f((s, x))$



=> 3+2+1 =6

=> +1 = 7 flow

3+1+1+2 =7

Minimum Cut

An $s - t$ **cut** is a partition of the nodes into sets $(A, V - A)$, where $s \in A$ and $t \in V - A$.

The **cutset** is the set of edges whose origin is in A and destination is in $V - A$.

The value of an $s - t$ cut is equal to the sum of the capacities of the edges in the cutset.

Problem Statement: Find the minimum cost $s - t$ cut.

1. What is the value of the mincut in the previous graph? 7

2. Anyone think this is a coincidence?

   not a coincidence

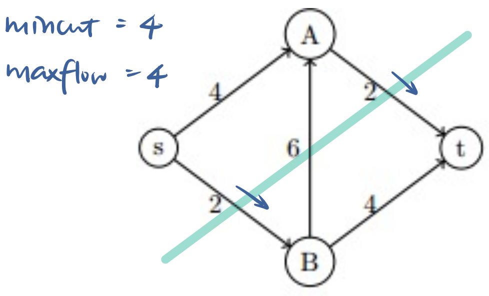
Weak Duality: Given a flow f and an $(A, V - A)$ cut, $v(f) \leq$ the value of the cut.

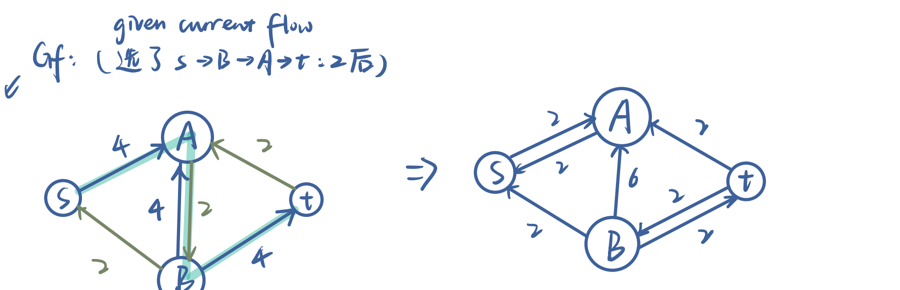
Corollary to Weak Duality: If $v(f) =$ the value of the cut, then f is a max-flow, and $(A, V - A)$ is a mincut. value of mincut 决定 max-flow 的上限

The remaining question is, can this always be done? Is there always a flow and cut with equal values?

Let's try to solve the max-flow problem using a greedy algorithm. We'll just choose arbitrary $s - t$ paths that have a remaining capacity, and route as much flow as possible along this path.

Given the original graph G, and the flow so far f, you can construct the residual graph G_f, which contains two types of edges.





到这儿停了，因为 no more s-t path

- Forward edges (u, v), which have capacity equal to the original capacity $c(u, v)$ minus the flow along the edge $f(u, v)$. $G_f(<u,v>) = c(<u,v>) - f(<u,v>)$
- Backward edges (v, u), which have capacity equal to the flow along the edge in the opposite direction $f(u, v)$. $G_f(<u,v>) = f(<v,u>)$

Forward edges indicate you can still augment the flow along that edge. Backward edges indicate that you can "take back" your choice to push flow along that edge and find a better solution.

Why was this a valid thing to do? Are the two rules of flow maintained if we push flow "backwards"? 主要是流量守恒. take steps back 对于点来说是 redirect input/output flow, 但数量不变

3→2

① The **Ford-Fulkerson** algorithm looks for an **augmenting path** on the residual graph (that is, a path from s to t where you can push more flow), pushes the maximum possible amount of flow along that path, and repeats until there are no more paths.

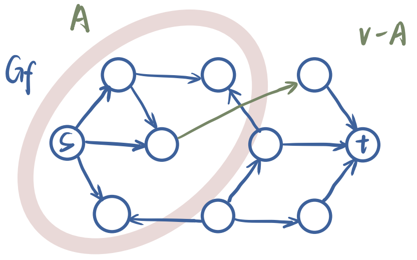
Augmenting Path Theorem: f is a max flow iff there are no s-t paths in G_f.

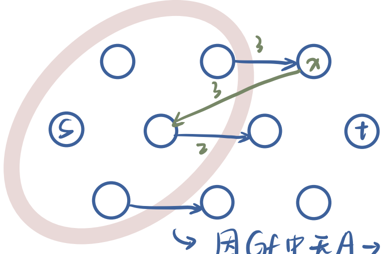
② **Max-flow Min-cut Theorem**: The value of the max flow equals the value of the mincut.

最大流最小割定理 2→1

都要证

[3→1 证明]:





不会出现反向从border回来的情况
不然Gf上应有对应的edge,
那x应∈A

加上这个
证明

↳ 因Gf中无A→V-A的edge了,
故这些cut都reach their capacity

We will prove the following three statements are equivalent:

1. There is a cut $(S, V - S)$ such that $v(f) =$ the capacity of the cut.
2. f is a max flow.
3. There is no s-t path in the residual graph G_f.

∴ value of cut = value of flow
⇒ max flow , min cut

1 → 2: We've already proven this.

2 → 3: Proof by contraposition. If there is an s-t path in G_f, then we can augment the flow, which means f is not a max flow. $\neg 3 \to \neg 2$, which means $2 \to 3$.

3 → 1: Let f be a flow with no augmenting path. Let A be the set of vertices reachable in G_f from s. Note that $s \in A$ and $t \in V - A$. This forms a cut of value equal to the flow!

Thus Ford-Fulkerson optimally solves Network Flow.

Algorithm for Min-Cut

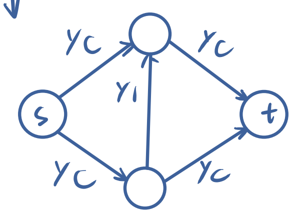
```
1: Run Ford-Fulkerson on the graph.
2: Create the resulting residual graph $G_f$
3: Let $A =$ the set of nodes reachable from $s$ in $G_f$
4: Return $(A, V - A)$
```

Integrality Theorem: If all capacities are integers, there is a max flow f where every value $f(e)$ is an integer. (因为每次都push max flow, 所以一直为整数)

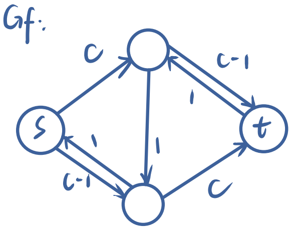
Let's analyze the running time of Ford-Fulkerson. The capacities of edges will be integers between 1 and C.

max capacity

1. Each time we do an augmentation, what is the minimum amount the total flow increases by? 1
2. How many total augmentations might we need? max flow ≤ nC
   For every s-t path, pushes ≥ 1 unit flow
3. How long should it take to find an augmenting path? (DFS) $\Theta(m+n) = \Theta(m)$ ← assume connected
4. What is the runtime? $\Theta(mnC)$ worst: repeat find s-t path and push one flow each time
   ⇒ $\Theta(m \cdot nC)$ → push nc times (⇒ find the path nc times)
   ↑ DFS each time
5. Is this a polynomial runtime? pseudo polynomial
6. Is it actually possible for the runtime to be this bad? Yes
7. Any ideas on how to improve our algorithm?
   ① 每次找 max-capacity s-t path (改版 Dijkstra)
   ② 每次找 s-t path w/ fewest edges (BFS)



Gf:



push 1 flow through the middle path each time
and untake that 1 flow from that path
repeat
⇒ $\Theta(m \cdot nC)$

Recap:

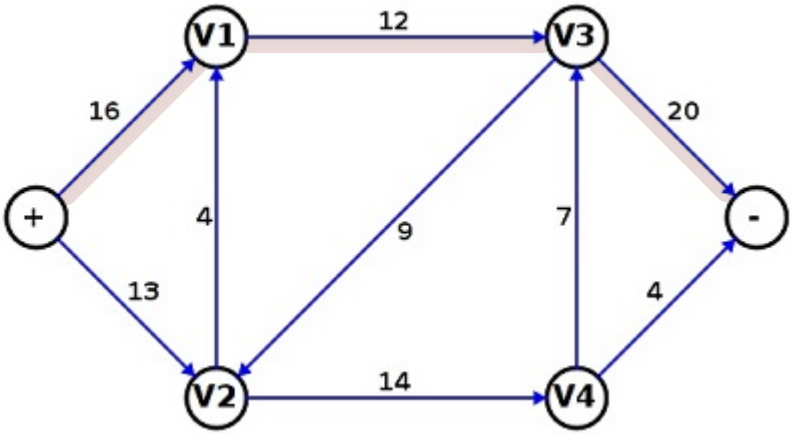
- Max-flow:
  - Min-cut (dual)
  - Ford-Fulkerson : runtime: $O(mnC)$ pseudo-poly
  - improvement? {
    - choose a path with max capacity
    - choose a path with fewest edge
    - choose a path with big enough capacity ⇒ capacity-scaling

Find a suitably-large s-t path

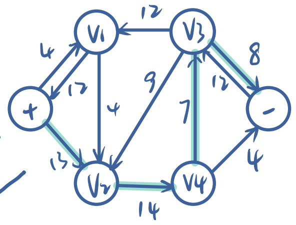
Capacity-Scaling

只用≥Δ的edge

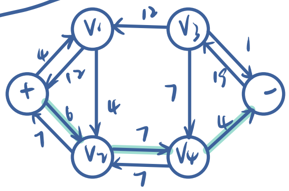
Let $G_f(\Delta)$ be the subgraph of G_f consisting only of edges with capacity $\geq \Delta$.

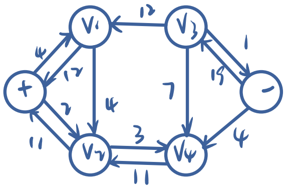


- What would $G_f(\Delta)$ look like for this graph if $\Delta = 16$? only 2 edge show up
- Are there any $s - t$ paths on $G_f(16)$? no
- What would $G_f(8)$ look like?
- Are there any $s - t$ paths on $G_f(8)$? yes
- What does G_f look like after we've augmented the flow?
- What does $G_f(8)$ look like at this point?
- What does $G_f(4)$ look like?



$12 + 7 + 4 = 23$





Ford-Fulkerson With Capacity-Scaling

repeat until $\Delta = 1$

runtime $= \log(C)$

```
1: Let $\Delta = 2^i$ for max $i$ such that $\Delta \leq C$.
2: While $\Delta \geq 1$
3:     Augment the current max flow using Ford-Fulkerson on $G_f(\Delta)$.
4:     $\Delta = \frac{\Delta}{2}$
```

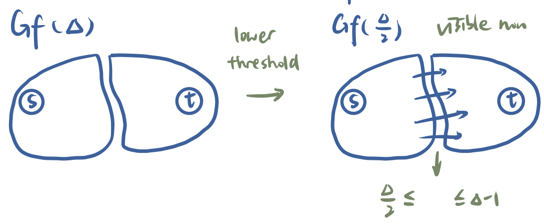
Now we will analyze the runtime of our algorithm.

We'll refer to each iteration as a phase. We need to figure out how many phases there will be, we need to figure out how long it takes to find a path on $G_f(\Delta)$, and we need to figure out how many paths we might find in each phase.

- How many iterations will our algorithm have?

  $\log_2 C$

At the end of Δ scale phase



max of C, # of edges

at most : $(\Delta - 1) \times m \approx \Delta m$ we can add

- each flow has capacity at least $\frac{\Delta}{2}$

- the most augmentation we need: $\frac{\Delta m}{\frac{\Delta}{2}} = 2m$

- How long does it take to find a path (with max flow) on $G_f(\Delta)$? Why? $2m \cdot m$ ↪ DFS

total runtime: $\log C \cdot 2m \cdot m$ $= \theta(m^2 \log C)$

So now we have to cap the number of paths we find in a phase.

Let f be the flow at the end of an iteration with Δ-scaling.

Let S be the set of nodes reachable from s in $G_f(\Delta)$.

Now consider G_f, which has additional edges of capacity $< \Delta$.

- Consider an arbitrary edge passing from S to V-S on G_f. What is the max amount of flow we could push along this edge?

- How many edges could pass from S to $V - S$? m

- How much flow could we conceivably be missing at this point in time? $m \cdot (\Delta - 1)$

Remember, we just finished the Δ-scaling phase. So in the next phase, we will consider $G_f(\frac{\Delta}{2})$.

- Suppose we find a new $s - t$ path when considering $G_f(\frac{\Delta}{2})$. What is the minimum amount of flow we will be able to push along this path? $\frac{\Delta}{2} \leq Gf(\frac{\Delta}{2}) \leq \Delta - 1$

- How much flow did we say we might be missing at this point in time? Δm

- How many paths can we find, maximum, during this phase? $\frac{(m-1)\Delta}{\frac{\Delta}{2}} = 2m$

Runtime for Ford-Fulkerson with Capacity-Scaling is $\theta(m^2 \log C)$. Is this a polynomial runtime? Yes total runtime: $\log C \cdot 2m \cdot m$ $= \theta(m^2 \log C)$ ⇒ polynomial

Other network flow algorithms:

- Preflow-Push $= O(n^3)$

- Goldberg-Rao $= O(min(n^{\frac{2}{3}}, m^{\frac{1}{2}})m \log n \log C)$

A lot of problems can be "reduced" to max-flow

(Lazy Algorithm)

CSCI 270 Lecture 21: Poly-Time Reductions

```
Minimum Cut(Graph $G$)
1: Find the max-flow $f$ and the residual graph $G_f$
2: Find the set of nodes $A$ reachable from $s$ on $G_f$
3: Return $A$
```

What is the runtime of Minimum Cut?

The runtime of Minimum Cut depends on the runtime of Maximum Flow! If someone finds a better Max-Flow algorithm, they will simultaneously improve the best known Min-Cut algorithm!

Min-Cut is not the only problem like this. There are a **lot** of problems for whom the runtime bottleneck is Max-Flow. This is why the research community has spent so much time trying to improve the existing Max-Flow algorithms.

This is called a **reduction**. In a reduction you take a problem you do not know how to solve, and turn it into a problem you do know how to solve.

In a **poly-time reduction**, the reduction takes no more than polynomial-time. This is also written: $MinimumCut \leq_p NetworkFlow$. Or: Minimum Cut is poly-time reducible to Network Flow.

$A =_p B$: A problem A can be reduced to another problem B in poly-time processing

- If $A \leq_p B$, and B is poly-time, what can we state about the running time for A?
  A can be solved poly-time
- If $A \leq_p B$, and A is poly-time, what can we state about the running time for B?
  Nothing can be said about B

So, by reducing Minimum Cut to Network Flow in polynomial time, we have proven that Minimum Cut is also solvable in polynomial time!

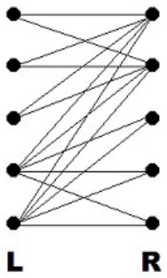
Now that we know Minimum Cut has a polynomial time algorithm, if we can give a poly-time reduction from some problem X to Min-Cut, we can determine that X has a polynomial-time algorithm as well.

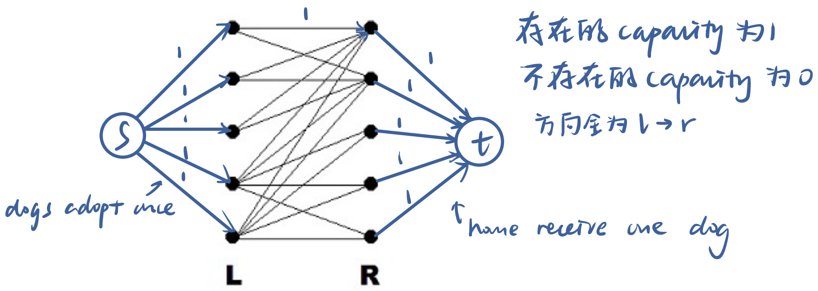
Let P be the set of problems with polynomial time solutions. If we want to show a problem is in P, we either write a poly-time algorithm for it, or we reduce it to a problem we already know is in P.

P np → none-polytime
↓
polytime

Bipartite Matching

We are given an undirected bipartite graph $G = (L \cup R, E)$.





存在的 capacity 为1
不存在的 capacity 为0
方向全为 l → r

$M \subseteq E$ is a matching if each node appears at most once in M. Find the max-size matching.

- What is the largest-sized matching you can find?
- Why can't there be a larger matching?

A perfect matching includes every node. This instance cannot have a perfect matching.

We could try to come up with our own algorithm to solve this problem, but that sounds like too much work. Instead we're going to be **lazy**, and use an existing algorithm. Namely, we'll use Ford-Fulkerson.

To use Ford-Fulkerson, we have to transform the current graph into a network flow graph. In addition, it must be done in such a way that the solution to the network flow graph helps us find the max-sized matching.

四步走：

- What components are missing in this graph, which are needed in a Network Flow graph? s, t, capacity

① transformation $\Theta(m+n)$

- How should we transform our graph into a Network Flow graph?

② extraction $\Theta(m)$

- How do we extract the answer to Bipartite Matching once we've solved Network Flow?
  = Max-flow from s to t

So, by reducing Bipartite Matching to Network Flow in polynomial time, we have proven that Bipartite Matching is also solvable in polynomial time!

Now that we've reduced Bipartite Matching to Network Flow, Bipartite Matching is a valid problem to reduce to as well. You could reduce some new problem $C \leq_p$ Bipartite Matching to show C is poly-time solvable.

③ poly-time reduction $\Theta(mnC) = \Theta(mn)$

一只狗只能 adopt 一次 s→dog c=1 ,

一户人只能养一只狗 home→t c=1

④ explanation : 1) if there is a flow of x, then there is matching of size x ⇒ iff 充要

2) if there is a matching of size , then there is a flow of x

↓ intuitive explanation :

Why this algorithm gains maximized matches?

Proof:

I. If there is a flow x in G', the edges with flow form a valid matching because each node can be passed by at most 1 unit of flow

II. if there is a match in G, there is a flow x in G'

不允许重复 node ⇒ 每个 node 只能在一个 path 上

又∵每个 node 只能在一个 pair 里

⇒ # of matching = maximum flow (# of path with weigh 1)

Design Template (Reduction to max-flow G)

1. Construct a network from a given graph (Eg: create node s and t, …)

2. Explain how to extract solution from an obtained max-flow (flow → matching)

3. Explain size x solution in max-flow implies size x solution in your problem   flow → matching

4. Explain size x solution in your problem implies size x solution in max flow   matching → flow

5. Explain the reduction takes polytime (E.g. bipartite matching : 加两个点 connect 全部点一遍)

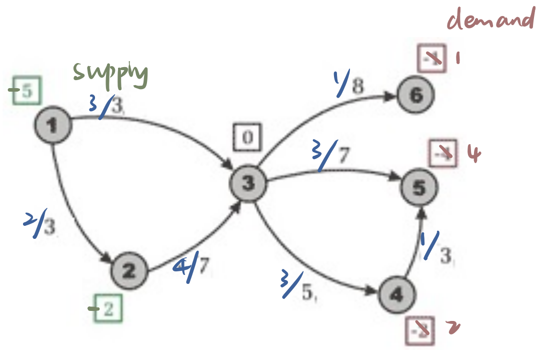
Circulations

We are given a directed graph G, with edge capacities $c(e)$, for all $c \in E$.

Each node has an integer **demand** $d(v)$. If $d(v) < 0$, then we say that this node has a **supply**.

A circulation is a function f which satisfies:

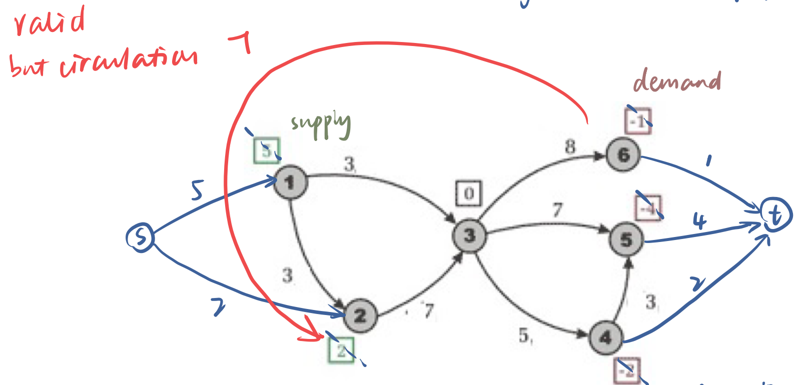
1. $0 \leq f(e) \leq c(e)$, for all e
2. $\sum_{(x,v)} f(x,v) - \sum_{(v,y)} f(v,y) = d(v)$



← valid answer

- Is there a circulation that satisfies these constraints? yes
- What components are missing in this graph, which would be needed in Network Flow?
- What components are in this graph, which must be removed in Network Flow?
- How should we transform our graph into a Network Flow graph?
- How can we extract the answer for Circulations, once we've solved Network Flow?

把 s, t 和相关的 edge 删去, 剩下的为所需的部分



transformation: $\Theta(n)$

ⓢ → supply nodes, c = supply

demand nodes → ⓣ, c = demand

extraction: $\Theta(m)$

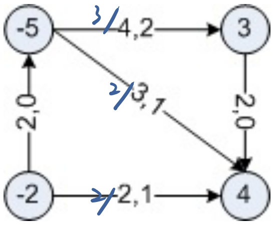
$\Theta(m) + \Theta(n)$ + poly-time → poly-time

if 所有从ⓢ出去的 capacity 都被用到了 ⇒ there is a circulation

和所有进ⓣ的 capacity

Circulations with Lower Bounds

Suppose edges have lower bounds $l(e)$. The new capacity rule is: $l(e) \leq f(e) \leq c(e)$



circulation of lowerbound $\leq_p$ circulations

所转换成的：① netflow ② min-cut ③ bipartite-matching ④ circulation ⑤ circulation with lower bounds

- Is there a circulation that satisfies these constraints?
- What problem should we reduce to? (Hint: NOT Network Flow!)
- What components are in this graph, which must be removed?
- How should we transform our graph?
- How can we extract the answer for Circulations with Lower Bounds?

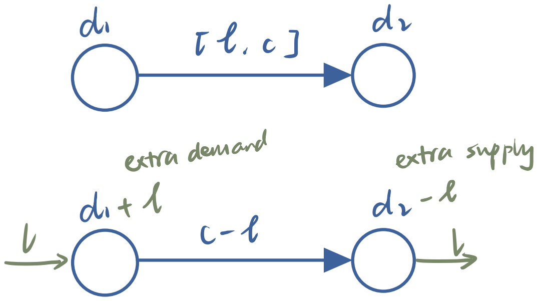
You may reduce to Network Flow, Bipartite Matching, or Circulations with or without lower bounds. You may reduce to other problems in P as well, but this is a unit on Network Flow, so it will be most fruitful to focus on those problems.

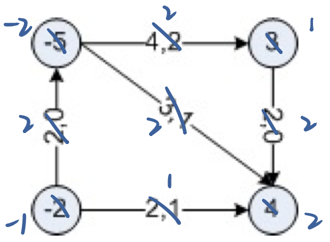
Important: don't mix up your problems. For example: Network-Flow with lower bounds is NOT a problem we have shown to be in P. You must do Circulations with lower bounds, or Network Flow without lower bounds.

Extra Problems:

- Chapter 7, exercises 8, 11, 12, 14, 24, 27, 29
- Challenge problems: Chapter 7, exercises 22, 41

然后解这个图的circulation





CSCI 270 Lecture 22: Poly-Time Reduction Examples

Survey Design

There are n customers $\{c_1, ..., c_n\}$ and m products $P = \{p_1, ..., p_m\}$. Customer c_i owns a subset of the products $S_i \subseteq P$.

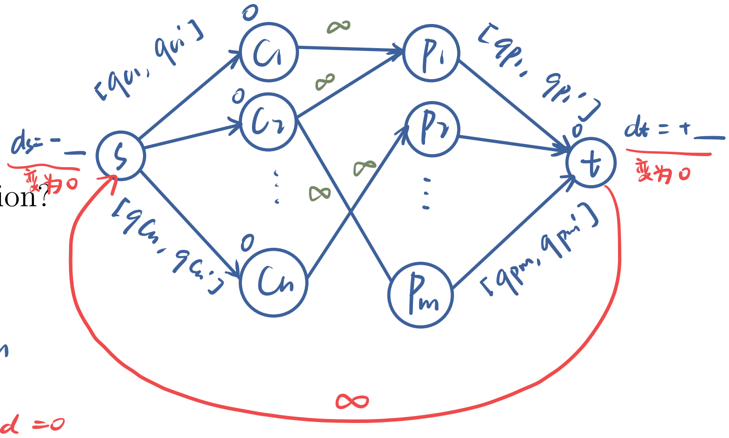
We want to design a survey which gets feedback on all of our products: we must ask at least qp_i questions (to get meaningful feedback) and no more than qp'_i questions (because the questions would become redundant) about product p_i, and we can only ask questions to customers which own p_i.

Lastly, we must ask at least cp_i questions to customer i (to avoid wasting their time) and no more than cp'_i questions (to keep the survey a reasonable length. Give a polynomial-time algorithm to design such a survey or find that no such survey exists.

→ 对Ci可问无数个关于Pj的问题

start by making a graph

- What problem should we reduce to?
- How should we transform survey design?
- Are there any problems with the transformation?
  不知道 s 和 t 的 demand
- How can we fix this problem?
  但 s 的 demand = t 的 supply



① 可+ edge from t to s → 转为 circulation with lower bound

s 和 t 的 demand =0

然后 flow on edge from t → s 就是# of question asked

② 可转为 max-flow
(把 lowerbound 化解掉)

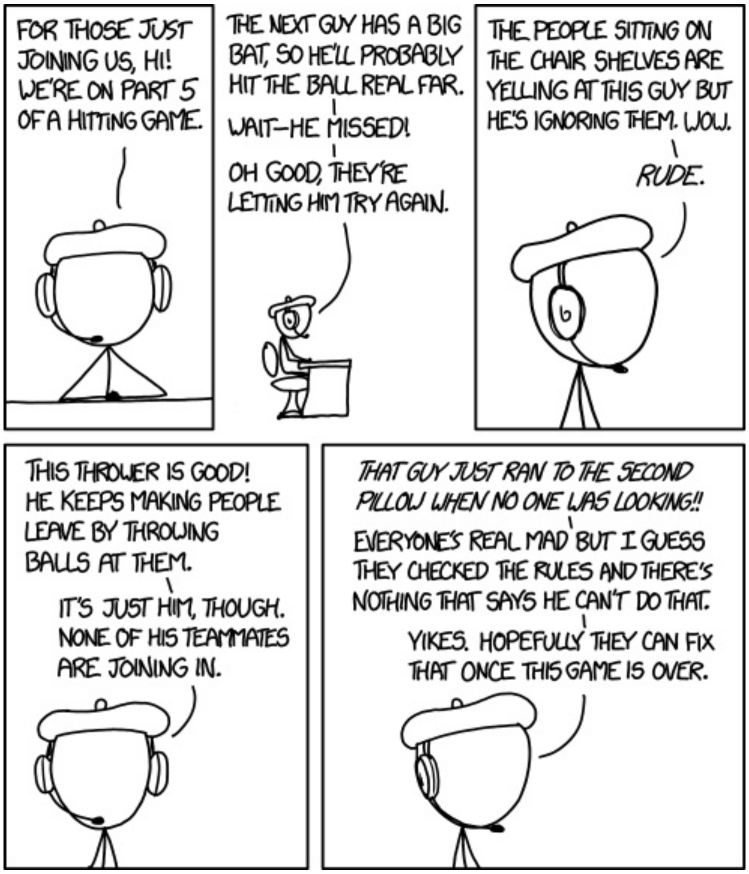

FOR THOSE JUST JOINING US, HI! WE'RE ON PART 5 OF A HITTING GAME.
THE NEXT GUY HAS A BIG BAT, SO HE'LL PROBABLY HIT THE BALL REAL FAR.
WAIT—HE MISSED!
OH GOOD, THEY'RE LETTING HIM TRY AGAIN.
THE PEOPLE SITTING ON THE CHAIR SHELVES ARE YELLING AT THIS GUY BUT HE'S IGNORING THEM. WOW.
RUDE.
THIS THROWER IS GOOD! HE KEEPS MAKING PEOPLE LEAVE BY THROWING BALLS AT THEM.
IT'S JUST HIM, THOUGH. NONE OF HIS TEAMMATES ARE JOINING IN.
THAT GUY JUST RAN TO THE SECOND PILLOW WHEN NO ONE WAS LOOKING!!
EVERYONE'S REAL MAD BUT I GUESS THEY CHECKED THE RULES AND THERE'S NOTHING THAT SAYS HE CAN'T DO THAT.
YIKES. HOPEFULLY THEY CAN FIX THAT ONCE THIS GAME IS OVER.

Figure 1: XKCD #1593 . The thrower started hitting the bats too much, so the king of the game told him to leave and brought out another thrower from thrower jail.

Baseball Elimination

Team	Wins	LA	Col	SF	Ari	SD
Los Angeles	90	0	0	0	7	4
Colorado	88	0	0	0	0	1
San Francisco	87	0	0	0	0	4
Arizona	86	7	0	0	0	4
San Diego	75	4	1	4	4	0

You want to determine if a team can end up with the most wins at the end of the season (or tied for most wins, forcing a playoff).

- Which teams are clearly eliminated?
- There is another team that is also mathematically eliminated: which one?

We want to design a poly-time algorithm that determines whether a specific team has been mathematically eliminated. For this example, let's consider San Francisco. It's not clear off-hand which problem to reduce to, but that's not an issue, because they're all graph problems: we can start by creating a graph and then decide which problem we're reducing to later. The flow will probably indicate who wins which games.

- What nodes should we add to this graph?
- Should we include San Francisco in our graph, or do we already know how to allocate their games?

We need to make sure that only teams involved in a matchup can win that specific game. We'll therefore add a node for each matchup.

- How should we finish our transformation?
- What problem are we reducing to?
- How do we extract the answer?

Project Selection

We have a set of n projects $P = \{p_1, ..., p_n\}$ with values $v_1, v_2, ..., v_n$ (the values may be negative).

(co-requisite)

Project i has a set of prerequisites $S_i \subseteq P$. 如果有 cycle, just do all nodes in the cycle

If we decide to do project i, we get value v_i, but we must also do all projects in S_i (and some of them may have negative values).

make all pre-requisite ∞ → invalid answer

⇒ 做后面不做 prerequisite 的 cut 是 ∞ （s → t 的 cut 是 ∞）

⇒ 连到 t 的为做了的 projects（这样从 t 到 s 的 edge 不算 cut

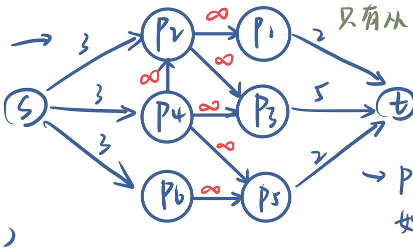
只有从 s 到 t 的才算 cut）

Find max-valued subset of projects to do.

$v_1 = 2, S_1 = P_2$
$v_2 = -3, S_2 = P_4$
$v_3 = 5, S_3 = P_2, P_4$
$v_4 = -3$
$v_5 = 2, S_5 = P_4, P_6$
$v_6 = -3$

s pointing to projects with negative values

（这样 cut 的时候，说明做了 project，⇒ lose points: +3 +3 +3）



做 P4 不做 P5 （t → s）无所谓

→ projects we want to do

如果不做：loss 增加

- What is the total value if we do all projects?
- What is the total value if we do no projects?
- Is there a better subset of projects?

only positive solution: P1, P2, P3, P4

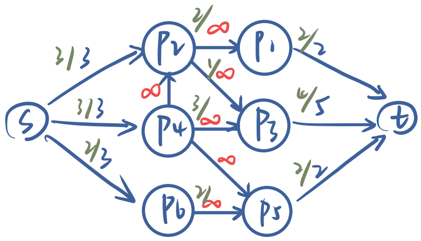
min-cut: optimization problem

⇒ partition problem

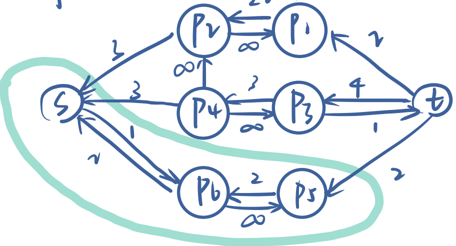
here:

2 parts: projects we do

projects we don't



Gf: (last residual graph: no more flow)



min-cut = 3+3+2

↓ = 8

离全正值之和的差

- We need to turn this into a graph. What should the nodes be?
- How should we encode the prerequisites?
- Would Bipartite Matching be a good problem to reduce to?
- Is there any problem with network flow?
- Would Circulations with Lower Bounds solve this problem? 不能，∵要给option可不做
- Are there any problems left?
- How do we make sure that we don't select a project without its prerequisites?
- Which set specifies the projects we do and which we don't do?
- We need to add edges from s to certain projects. If we split one of these edges, what does that say about the project?
- We need to add edges from certain projects to t. If we split one of these edges, what does that say about the project?
- We need to add the appropriate penalties to completing a project with negative value, and appropriate incentives to completing a project with positive value. How should we go about doing this?

Proof of correctness:

For every project with negative value in S, we increase the value of the cut:

$\sum_{i \in S, p_i < 0} -p_i$

We find a min-cut $(S, V - S)$. For every project with positive value in $V - S$, we increase the value of the cut:

$\sum_{i \in V-S, p_i > 0} p_i$

Therefore the value of the cut we find is

$\sum_{i \in V-S, p_i > 0} p_i + \sum_{i \in S, p_i < 0} -p_i$

Let $C = \sum_{i: p_i > 0} p_i$: the sum of capacities of edges outgoing from the source. Then:

$\sum_{i \in V-S, p_i > 0} p_i = C - \sum_{i \in S, p_i > 0} p_i$

Then the total value of our cut is the sum of these two values:

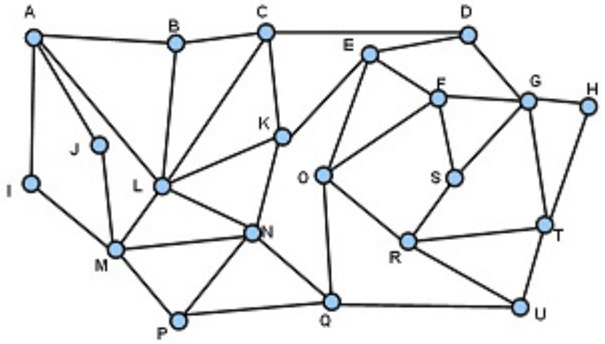
$C - \sum_{i \in S, p_i > 0} p_i + \sum_{i \in S, p_i < 0} -p_i = C - \sum_{i \in S} p_i.$

Since $\sum_{i \in S} p_i$ is exactly our profit, if we minimize $C - \sum_{i \in S} p_i$ then we maximize our profit. Thus, finding min-cut here really works!

CSCI 270 Lecture 23: The Limits of Knowledge

Independent Set (max)

Given a graph G and an integer k, is there a set of nodes $S \subseteq V$ such that $|S| \geq k$, and there are no edges between two nodes in S?



$VC \leq_p IS$

$IS \leq_p VC$

Vertex Cover (smallest)

Given a graph G and an integer k, is there a set of nodes $S \subseteq V$ such that $|S| \leq k$ and every edge has at least one endpoint in S?

```
VC(G){
    G→sets
    ans = SC(sets)
    ans→ans'
    return ans'
}
```

If we have an algorithm for SC, we would have an algorithm for VC

Since we don't have an algorithm for SC, we don't have an algorithm for VC

Set Cover

We are given a set U of n elements, and m subsets $S_1, S_2, ..., S_m \subseteq U$. Given an integer k, is there a collection of $\leq k$ subsets whose union is equal to U?

$U = \{1, 2, 3, 4, 5, 6\}$

$S_1 = \{1, 2\}, S_2 = \{2, 3, 5\}, S_3 = \{2, 4\}, S_4 = \{4, 5\}, S_5 = \{3, 6\}, S_6 = \{5, 6\}$

we want to show set cover is hard by showing $VC \leq_p SC$ ★ node → set edge → element

The class NP ★ turn node into set ← (解VC时用SC)

choosing the set cover U

⇒ choosing the node cover all edges

Suppose you have an Independent Set example which you have been unable to solve. Albert Einstein walks in, looks at it, and say "Yes! There is an Independent Set of size k, there it is!"

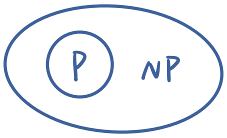
How long would it take you to verify that Albert Einstein's answer is correct?

Suppose Einstein did the same for a Vertex Cover or Set Cover problem. How long would it take you to verify his solution is correct? $O(n)$

A verifier/certifier is like a grader. It does not come up with a solution on its own. It just verifies a given solution is correct.

P is the set of all problems with polynomial-time solutions.

NP is the set of all problems with polynomial-time VERIFIERS.



P=Polynomial Time

NP=Nondeterministic Polynomial Time

CS 475

Effectively we are saying "If we could guess the solution, we could verify its authenticity in polynomial time".

- What problems have we seen that are in NP?
- Is Sorting in NP? yes
- Is $P \subseteq NP$? yes
- Are there problems which are not in NP?
- Is $P \underset{\subsetneq}{\subset} NP$? mystery (we think 99% true)

Alright, let's suppose we want to prove it one way or the other. Here is a possible strategy:

1. Identify the "hardest" problem in NP.
2. Either give a polynomial algorithm for it, or prove no polynomial algorithm is possible.

It's not clear that there is a "hardest" problem in NP, but we should be able to formally define what it **means** to be the hardest problem in NP. What kind of traits do you suppose such a problem would have?

$\forall z \in NP$, $z \leq_p$ hardest problem

(everything is reducible to the hardest problem)

Cook-Levin Theorem early 70's

Figure 1: Futurama, depicting P≠NP.

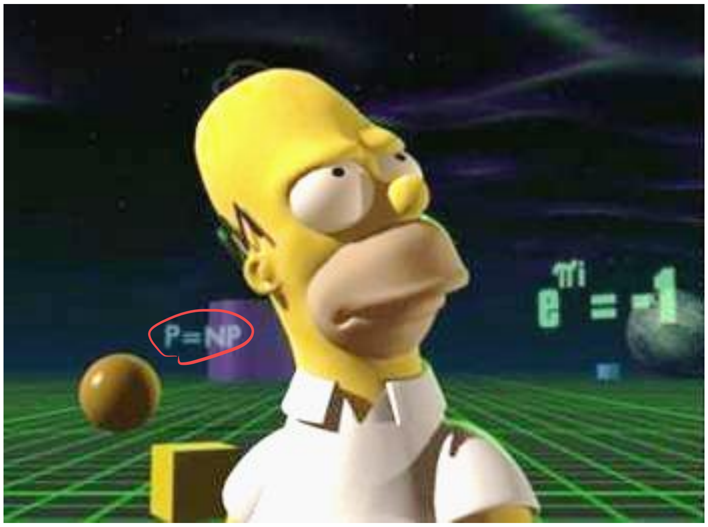


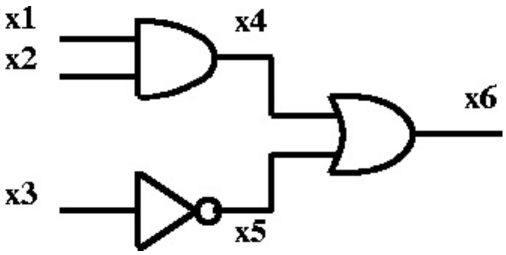
Figure 2: The Simpsons, depicting P=NP.

Circuit-SAT

Given a combinational circuit of AND, OR, and NOT gates, is there a way to set the inputs such that the output is 1?

Is Circuit-SAT $\in$ NP?

Claim: For any problem $x \in$ NP, $x \leq_p$ Circuit-SAT.



- A Turing Machine is a simple mathematical model of a computer. Anything a computer can do, a TM can do (but slowly). TMs are not real things, they are a theoretical concept. Because they are simple, we can formally prove what kinds of things computers can and cannot do.
- A nondeterministic computer is another theoretical concept which does not actually exist. It is effectively an infinitely parallel computer which can try all possible answers simultaneously. Thus, any problem in NP can be solved by a NTM in poly-time, because all it has to do is verify every possible answer in parallel.
- Computers are just circuits. It stands to reason then that we could take a TM and translate it into a circuit which contains the exact same logic.

Effectively: any problem in NP can be reduced to Circuit-SAT in poly-time.

Circuit-SAT is what is called an NP-complete problem:

1. Circuit-SAT $\in$ NP
2. For all $x \in$ NP, $x \leq_P$ Circuit-SAT

Suppose we have another problem Y, and we show:

1. Y $\in$ NP
2. Circuit-SAT $\leq_p$ Y (假设已有Y的解, 用其解出 Circuit-SAT)

What have we shown? Y is also NP-complete problem

Suppose we have another problem Z, and we show:

1. Z $\in$ NP
2. Z $\leq_P$ Circuit-SAT

What have we shown?

CSCI 270 Lecture 24: NP-Completeness

3-SAT

We have a set of n variables $x_1, x_2, ..., x_n$, and a set of m clauses $C_1, C_2, ..., C_m$. Each clause is a disjunction of exactly 3 variables or their negation, such as $C_1 = (x_1 \vee \neg x_2 \vee x_4)$.

CNF (conjuctive normal forms) $(\vee^- \vee^- \vee) \wedge (\vee \vee) \wedge (\vee \vee)$

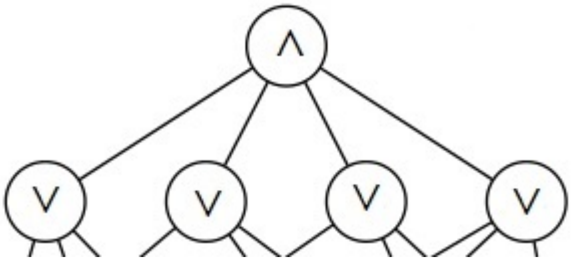
A 3-SAT formula is a conjunction of all clauses $C_1 \wedge C_2 \wedge ... \wedge C_m$, such as:
$(x_1 \vee \neg x_2 \vee x_4) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$.

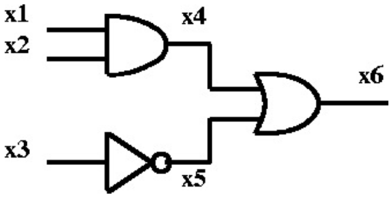
We want to know if there is an assignment of boolean values to variables such that the formula evaluates to true.

- Is there a satisfying assignment for this example?
- Is 3-SAT $\in NP$? yes (run through the formula to verify)
- What's wrong with the following reduction? circuit$_{SAT}$ $\leq_p$ 3-SAT

应该用 3-SAT 解 circuit$_{SAT}$

把 circuit$_{SAT}$ 转成 3-SAT 的问题





Reduction from Circuit-SAT to 3-SAT:

not-gate:
$x_5 \equiv \neg x_3$
$(x_3 \wedge \neg x_5) \vee (\neg x_3 \wedge x_5)$
$\Rightarrow (x_3 \vee x_5) \wedge (\neg x_3 \vee \neg x_5)$

1. Add variable names to every wire in the Circuit-SAT problem.
   → 让 3-SAT 的结果和 circuit-SAT 的一致
2. Hard-code output: $C_1 = (x_6)$ $= (x_6 \vee x_6 \vee x_6)$
3. Transform not-gates. $x_5 = \neg x_3$ becomes $C_2 = (x_3 \vee x_5)$ $\vee x_5$ and $C_3 = (\neg x_3 \vee \neg x_5)$ $\vee \neg x_5$

translate: each gate exists $\Leftrightarrow$ 转为的clauses均为true

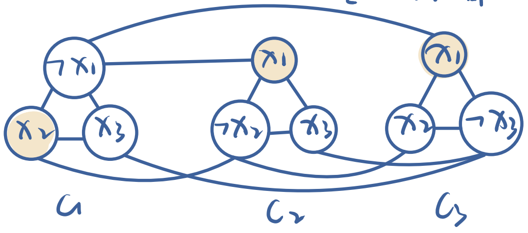
   $x_4 \Rightarrow x_6$ $x_5 \Rightarrow x_6$
4. Transform or-gates. $x_6 = x_4 \vee x_5$ becomes $C_4 = (x_6 \vee \neg x_4)$ and $C_5 = (x_6 \vee \neg x_5)$ and $C_6 = (\neg x_6 \vee x_4 \vee x_5)$ ↳ $(x_6 \wedge (x_4 \vee x_5)) \vee (\neg x_6 \wedge \neg x_4 \wedge \neg x_5)$
   $(\neg x_4 \wedge \neg x_5) \Rightarrow \neg x_6$
5. Transform and-gates. $x_4 = x_1 \wedge x_2$ becomes $C_7 = (\neg x_4 \vee x_1)$ and $C_8 = (\neg x_4 \vee x_2)$ and $C_9 = (x_4 \vee \neg x_1 \vee \neg x_2)$ ↳ $\neg x_1 \Rightarrow \neg x_4$
   $\neg x_2 \Rightarrow \neg x_4$
6. What's missing in our reduction? $x_1 \wedge x_2 \Rightarrow x_4$

Independent Set

translate logic formula into a graph

We know IS $\in$ NP. Now we want to show that 3-SAT $\leq_p$ IS. Turn an arbitrary 3-SAT instance into an IS problem.

$(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$



← 不让IS都选

1. Add 3 nodes in a triangle for each clause.
2. What's missing in our reduction?
3. How many problems have we shown are NP-Complete?

we need to find one node per clause

IS解的全集可 ⇒ 3-SAT 解的全集 (∵未被选到就是都可选)

Remember, this example is only an illustration to clarify my proof. My proof must work on all 3-SAT instances, not just this one!

There are a lot of NP-complete problems. This is why we highly suspect that P $\neq$ NP. All we would have to do to prove P=NP is come up with a polynomial time algorithm for one of these problems. With how much effort has been expended, we probably would have done so by now if it were possible. Proving that no polynomial algorithm exists for a problem is much much harder.

Set Packing

Given n elements $U = \{u_1, u_2, ..., u_n\}$, m subsets $S_1, S_2, ..., S_m \subseteq U$, and an integer k. Are there k sets which don't intersect?

Think about the similarities between the problems. In Independent Set we are packing as many nodes as we can such that no edge is represented twice. In Set Packing we are packing as many subsets as we can such that no element is represented twice.

k-Clique

Given a graph and an integer k, is there a set S of $\geq k$ nodes such that every pair of nodes in S have an edge between them?

There are a lot of NP-complete problems. This is why we highly suspect that P $\neq$ NP. All we would have to do to prove P=NP is come up with a polynomial time algorithm for one of these problems. With how much effort has been expended, we probably would have done so by now if it were possible. Proving that no polynomial algorithm exists for a problem is much much harder. ☆ $\in$ NP

IS $\leq_p$ k-clique (∵IS 求所有无edge的点, k-clique 求有所有edge的点)

invent all edges of IS, find k-clique ans for the new graph

get the ans for IS

一定要show $\in$ NP ⇒ NP complete

CSCI 270 Lecture 25: Sequencing Problems

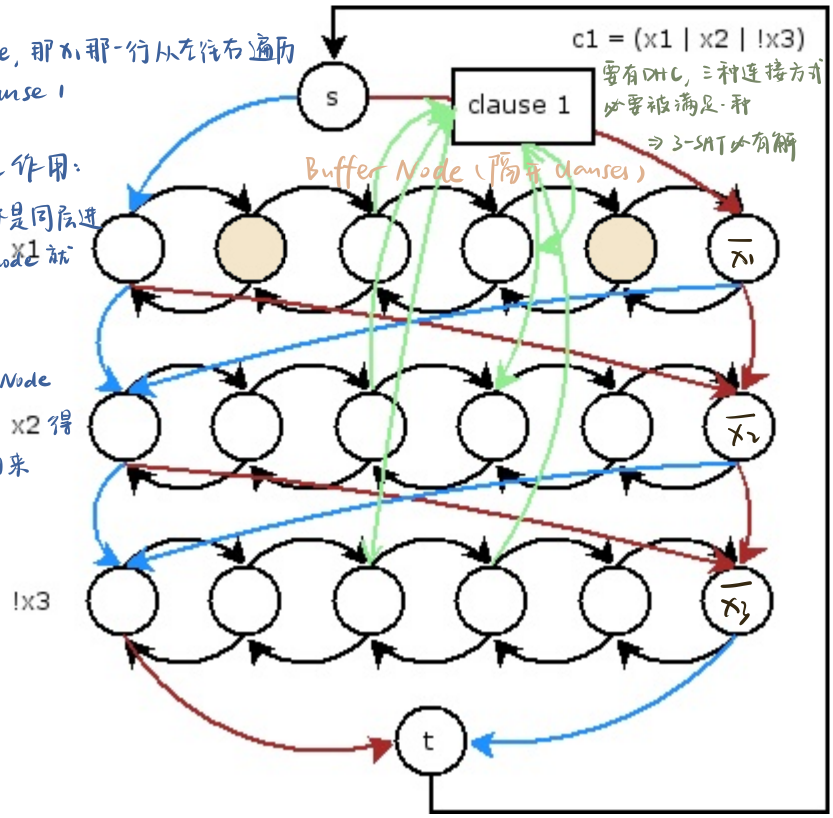
(DHC)

Directed Hamiltonian Cycle ∈ NP

Given a directed graph $G = (V, E)$, is there a simple directed cycle C that visits every node?

如果x1为true，那x1那一行从左往右遍历
必能路过clause 1

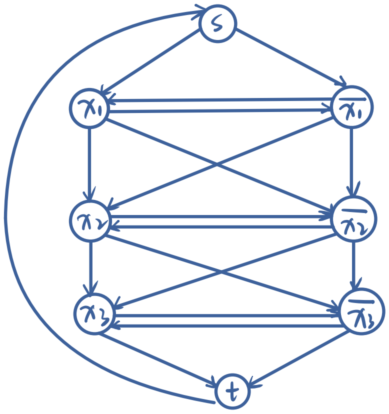
Buffer Node作用：
如果 clauses 不是同层进
同层出，buffer node 就
不会被满足
⇒ DHC要有解
⇒ 要过 Buffer Node
⇒ 遍历 clause 得
按我们规则来



Buffer Node (隔开 clauses)

要有DHC，三种连接方式
必要被满足一种
⇒ 3-SAT必有解

证 NP-complete：
1) DHC ∈ NP
2) 3-SAT $\le_p$ DHC



Undirected Hamiltonian Cycle

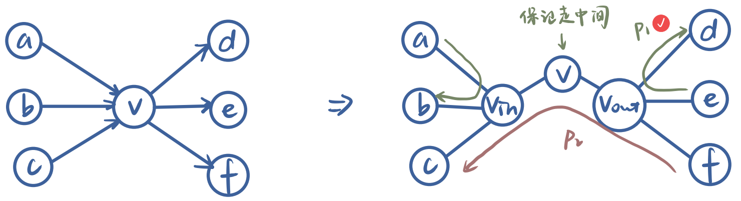
Given an undirected graph $G = (V, E)$, is there a simple cycle C that visits every node?

不写扣分 ☆

第一步：UHC ∈ NP

第二步：DHC $\le_p$ UHC

Turn Directed to Undirected :



把每个点都split成三个

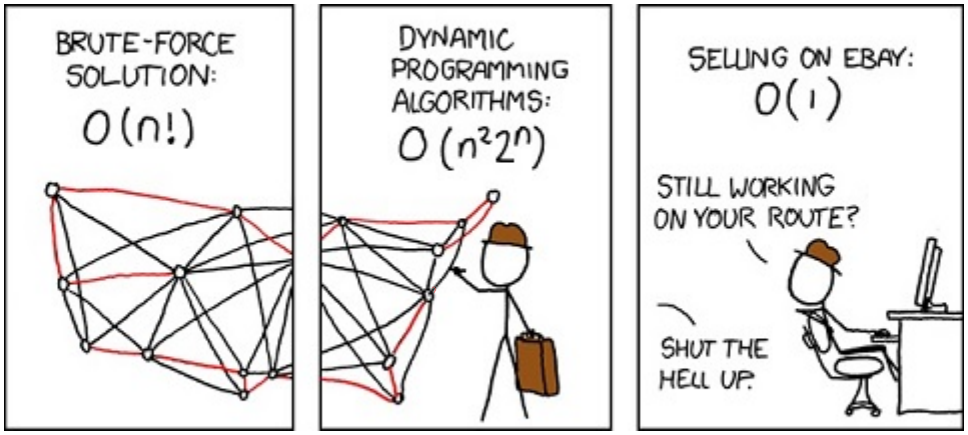
For problem 2 : 如果从 out 进 in 出
那整张图都是这样的
整张flip就行

Travelling Salesman Problem

Find Undirected Hamiltonian Cycle of weight ≤D

Given a set of n cities, a distance function $d(u, v)$ which specifies the distance between any two cities u and v, and a value D, find a tour of length $\leq D$.

From the comic XKCD:



注:

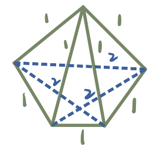
$d(a,b) + d(b,c) \geq d(a,c)$

每个cycle 满足三角形特形

(两边之和大于第三边)

TSP ∈ NP

UHC $\leq_p$ TSP



D = n

for triangle property

不存在的设为2

因为要n之内,

∴不能走 weight=2 的路

=> TSP ∈ NP complete

Directed Longest Path

Given a directed graph and an integer k, is there a path of $\geq k$ nodes?

Undirected Longest Path

Given an undirected graph and an integer k, is there a path of $\geq k$ nodes?

What makes a problem NP-complete?

How does one recognize an NP-complete problem? You can't until you give a reduction. It is very difficult to tell at a glance.

Longest path is NP-complete.

Longest path on a DAG is easy!

3-SAT is NP-complete.

2-SAT is easy!

Independent Set is NP-Complete.

Independent Set on a tree is easy!

Packing : maximizing problem

NP-complete → IS , k-clique , Set packing

Covering : make sure everything is represented [min problem (最小化选择)]

VC , SC

Constraint Satisfaction

circuit-SAT , 3-SAT

Sequencing : (order)

directed Hamiltonian Cycle

Numerical : add up to some value

subset sum

Partitioning

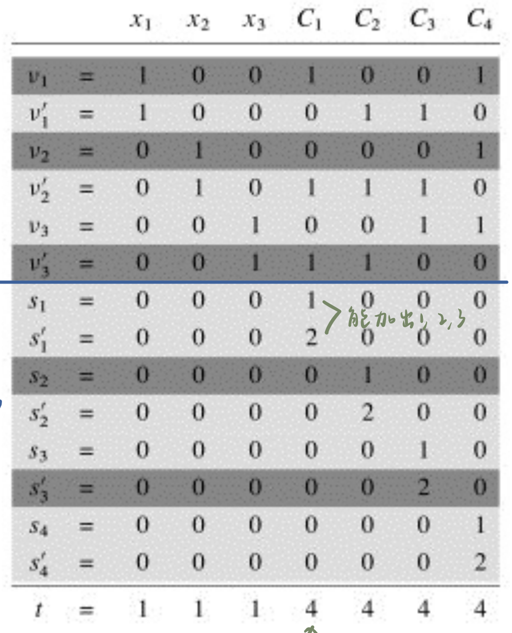
3-color

CSCI 270 Lecture 26: Numerical and Partitioning Problems

Subset Sum (SS)

DP: pseudo-polynomial

Given n positive integers $w_1, w_2, ..., w_n$ and a target W is there a subset of integers which add up exactly to W?



			x_1	x_2	x_3	C_1	C_2	C_3	C_4
	v_1	=	1	0	0	1	0	0	1
	v'_1	=	1	0	0	0	1	1	0
	v_2	=	0	1	0	0	0	0	1
	v'_2	=	0	1	0	1	1	1	0
	v_3	=	0	0	1	0	0	1	1
	v'_3	=	0	0	1	1	1	0	0
C_1	s_1	=	0	0	0	1	0	0	0
C_1'	s'_1	=	0	0	0	2	0	0	0
C_2	s_2	=	0	0	0	0	1	0	0
C_2'	s'_2	=	0	0	0	0	2	0	0
C_3	s_3	=	0	0	0	0	0	1	0
C_3'	s'_3	=	0	0	0	0	0	2	0
C_4	s_4	=	0	0	0	0	0	0	1
C_4'	s'_4	=	0	0	0	0	0	0	2
	t	=	1	1	1	4	4	4	4

Buffer to satisfy ④

能加出1, 2, 3

Buffer = C1, C1' 为了凑4
x1, x2, x3 满足一个 +1 +2
两 +2
三 +1

To prove SS ∈ NP-Complete

1) SS ∈ NP

2) 3-SAT $\le_p$ SS

m=4

$\overset{C_1}{(x_1 \vee \overline{x_2} \vee \overline{x_3})} \wedge \overset{C_2}{(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})} \wedge \overset{C_3}{(\overline{x_1} \vee \overline{x_2} \vee x_3)} \wedge \overset{C_4}{(x_1 \vee x_2 \vee x_3)}$

create m+2m numbers

each number will be n+m digits

	x_1	x_2	x_3	C_1	C_2	C_3	C_4
x_1	1	0	0	1	0	0	1
$\overline{x_1}$	1	0	0	0	1	1	0
x_2	0	1	0	0	0	0	1
$\overline{x_2}$	0	1	0	1	1	1	0
x_3	0	0	1	0	0	1	1
$\overline{x_3}$	0	0	1	1	1	0	0
W	1	1	1	1-3	1-3	1-3	1-3

⇒ 3^m (不够好)

✓ 3-SAT有解 ⟺ SS有解

3-Color

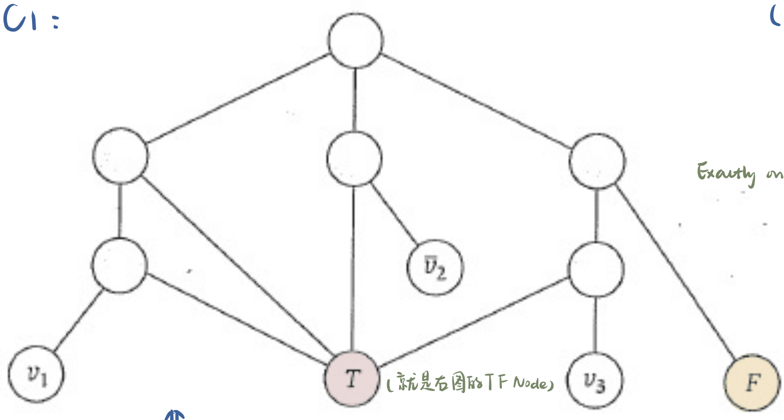
Given an undirected graph $G = (V, E)$, is there a way to assign one of 3 colors Red, Green, and Blue, to each node, so that no two adjacent nodes have the same color?

3-Color ∈ NP

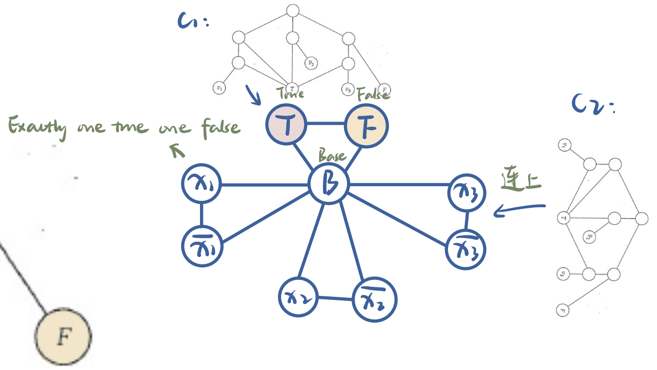
3-SAT $\le_p$ 3-color

C1:

$(x_1 \vee \overline{x_2} \vee x_3) \wedge \cdots$



(就是右图的T F Node)

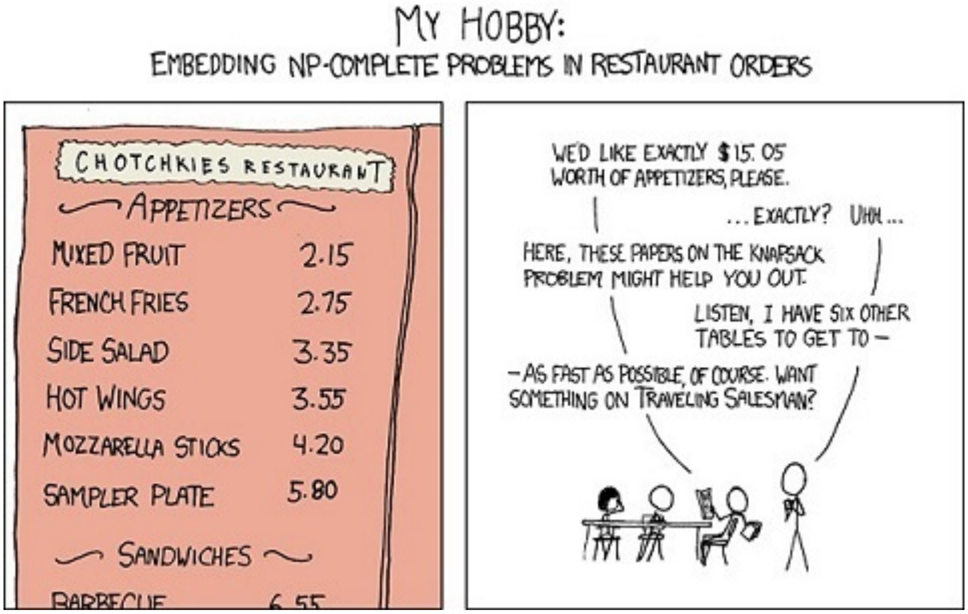


⇑

如果能在图上3-Color，那么3-SAT有解
(3 False 无解)

4-Color

Given an undirected graph $G = (V, E)$, is there a way to assign one of 4 colors Red, Green, Blue, and Purple to each node, so that no two adjacent nodes have the same color?


MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
BARBECUE 6.55
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO –
– AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

3-D Matching

Given n instructors, n courses, n times, and a list of 3-tuples listing valid pairings of courses, times, and instructors, find an assignment where all instructors teach, and all classes are taught at different times.

(Aaron, 170, MW 2-3:20), (Aaron, 170, TTh 11-12:20), (Aaron, 270, TTh 11-12:20)
(Tian, 104, MW 2-3:20), (Tian, 170, TTh 11-12:20)
(Brendan, 104, TTh 2-3:20), (Brendan, 170, TTh 2-3:20), (Brendan, 270, MW 2-3:20)

CSCI 270 Lecture 27: The Limits of Computation

you can't know

Suppose you are given some computer code, and the input which the code will receive. You want to know whether the code will halt, or enter an infinite loop.

Compilers can catch some very simple examples, but they can't catch the more complicated examples. It would be great if someone clever enough came along and wrote a program which could solve this problem. Why do you suppose no one has done this?

This problem is referred to as the **Halting Problem**.

The Barber Paradox

There is a barber in a small village. The barber cuts the hair of exactly the villagers who do not cut their own hair.

- What is the paradox? who cuts the barber's hair?
- Is this really a paradox?

An algorithm that solves the Halting Problem is the "barber". If we assume such an algorithm exists, we reach a paradox. The only conclusion is that such an algorithm cannot possibly exist.

Countable versus Uncountable

Consider two sets of numbers: the set of natural numbers $\{1, 2, 3, ...\}$, and the set of even natural numbers $\{2, 4, 6, ...\}$. Which set is larger? $f(x) = 2x$

Given two sets of infinite size, A and B, if there is a bijective function f which maps A to B, then these two sets are the same size.

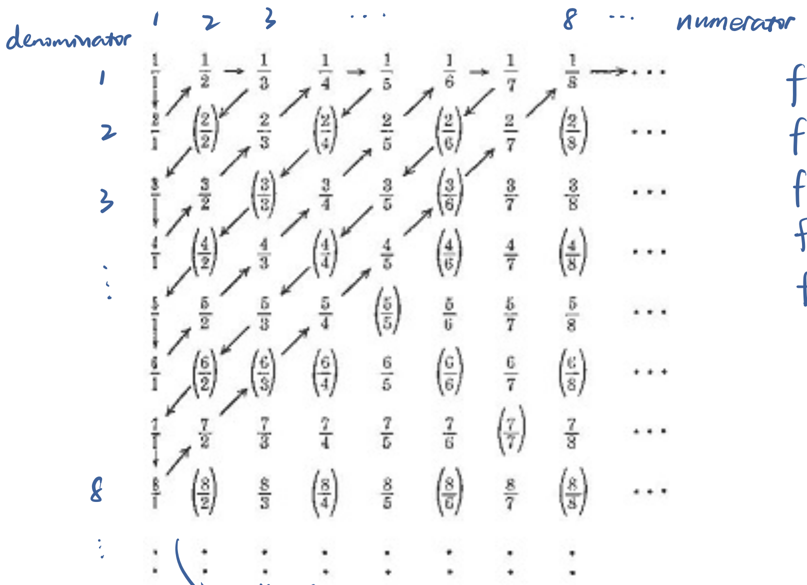
In less formal terms, if you can pair each number in A with a unique number in B such that every number in B is paired exactly once, then A and B have the same size. Any infinite set of size equal to the natural numbers is called "countable".

uncountable : if size > natural numbers

- Are the set of even integers countable? Yes
- Are the set of positive rational numbers countable? Yes
- Are the set of real numbers countable? No

Algorithms are countable

Problems are uncountable (> algorithms)



$f(0) = \frac{1}{1}$ ↙ algorithmatically countable

$f(1) = \frac{2}{1}$

$f(2) = \frac{1}{2}$

$f(3) = \frac{1}{3}$ (skip over repeats)

$f(4) = \frac{3}{1}$

⋮

这样写是因为如果 row by row (one row won't end)

Proof by contradiction: assume we have a bijective function f which maps A (the set of natural numbers) to B (the set of real numbers). It might be (for example) that:

irrational number (无限小数)

b 和 f 中的每个数都有一位小数不同

(Barber's)

- $f(1) = \pi$ 3.1415926
- $f(2) = e$ 2.71828
- $f(3) = 0.123456789$
- $f(4) = 32.\overline{33}$ (repeating, of course), etc.

Now we will construct a real number $b < 1$ which does not appear in this table, according to the following process (the process is referred to as *diagonalization*).

If the first decimal place of $f(1)$ is 1, then the first decimal place of b will be 2. Otherwise, the first decimal place of b will be 1.

If the ith decimal place of $f(i)$ is 1, then the ith decimal place of b will be 2, otherwise it will be 1.

Given the above example function, $b = 0.2211...$

- Does any input produce b as output? No (因为每个 input 都有一位小数不同)
- What does this have to do with Computer Science?
- Are the set of computer programs countable or not? Yes
  computer programming ⇒ string ⇒ Base 26 integers ⇒ countable
- Are the set of problems countable or not? No

To answer the last question, lets make some simplifying assumptions. All problems require as input a single natural number (clearly there are other problems which don't meet this criteria). In addition, all problems will return a boolean value (true or false).

Def:

Restricted Def for problems (已经不是全集了)

	1	2	3	4	...
M_0	F	F	F	F	...
M_1	T	T	T	T	...
M_2	F	T	F	T	...
⋮		⋮			

Define b by flipping diagonals

⇒ b is not included in f

⇒ problems are uncountable

Since the set of computer programs is countable, there is a bijective function from the natural numbers to the set of computer programs. We'll refer to the computer program which is output from $f(i)$ as M_i.

Now list out the computer programs in order along the x axis. At entry (i, j), we write what M_j outputs when it is run on the input i.

Can you identify a program which does not exist on this table?

Undecidable Problems

We will assume that the Halting Problem can be solved by program $H(M, w)$. It takes as input computer code M, and input to that computer code w and returns True, meaning that M halts on (M(w) = true) input w, or False, meaning that it doesn't halt. H always halts (otherwise it doesn't really solve the problem).

Using our solution to the Halting Problem, we can write a different program $B(M, w)$. It works as follows:

```
B(computer code M, input w)
  If (H(M,w)==False) Then Return True
  Else loop forever
```

构造 contradiction

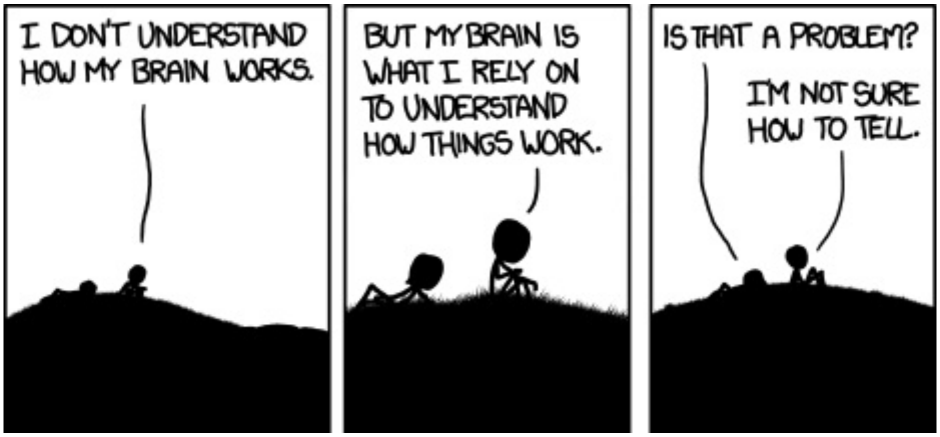
What happens when I run B on input $M = B$?

- If B(B,w) halts, then H(B,w) returns True, which means B(B,w) loops forever. Contradiction.
- If B(B,w) doesn't halt, then H(B,w) returns False, when means B(B,w) returns True. Contradiction.

Halting problem: 重要的是查自己!

B is the barber from the barber paradox. If you assume B exists, you get a paradox. The only possible conclusion is that B does not exist. We also showed $B \leq H$, so H doesn't exist either!

XKCD:


I DON'T UNDERSTAND HOW MY BRAIN WORKS.
BUT MY BRAIN IS WHAT I RELY ON TO UNDERSTAND HOW THINGS WORK.
IS THAT A PROBLEM?
I'M NOT SURE HOW TO TELL.