CS3230 Cheatsheet
Useful math stuff
- n! >= $(n/e)^n$
harmonic series:
1 + 1/2 + ... + 1/k ~=~ lnk.

$$\frac{d}{dx}\ln(x) = \frac{1}{x}$$

$$\frac{d}{dx}\log_b(x) = \frac{1}{\ln(b)\cdot x}$$

Logarithm Rules and Tricks summary

1. $\log_a(xy) = \log_a(x) + \log_a(y)$
2. $\log_a(x/y) = \log_a(x) - \log_a(y)$
3. $\log_a(x^n) = n\log_a(x)$
4. $\log_a(1) = 0$
5. $\log_a(a) = 1$
6. $a^{\log_a(n)} = n$
7. $a^{\log_b(c)} = c^{\log_b(a)}$
8. $\log_b(a) = \log_d(a) / \log_d(b)$

Arithmetic Progression

$$\sum_{i=1}^{n} i = (n(n+1))/2 = (n^2+n)/2$$ (Dividing both sides by 2)

Geometric Progression

$$\sum_{i=0}^{n-1} ar^i = \frac{(a - ar^n)}{(1-r)} = \frac{a(1-r^n)}{(1-r)}$$

Sum to Infinity:

$$= \frac{a}{(1-r)} = c, \text{ where } c \text{ is a constant}$$

Pigeonhole Principle
If we have n pigeonholes and n+1 pigeons, then one pigeonhole must contain more than 1 pigeon

Intro to Algorithms
- Running time measured by number of steps as a function of input size, what each step is depends on computational model.

Comparison Model: Maximum

- Problem: Given an array A of n distinct elements (denoted by $A_1, \ldots, A_n$), find the largest element in A.
- Here is one algorithm:

```
RunThru(A)
1  cur = 1
2  n = A.length
3  for i = 2 to n
4       if A_i > A_cur
5            cur = i
6  return A_cur
```

- RunThru makes exactly $n-1$ comparisons for any input.

Claim: Every algorithm solving the max problem must make >= n-1 comparisons
→ Adversary argument, if we make less than n-1 comparisons, then there exists 2 inputs which is indistinguishable to the algorithm M, but an adversary can adjust the inputs such that there are 2 different results. So it must be that at least n-1 comparisons must be made

1. To use adversary argument, we always start with the assumption that the algorithm makes less than x number of comparisons/queries (x is the claim of the number of comparisons we need). If is query, define what the query is and what the adversary replies
2. Define what the adversary will do
3. Show that the algorithm cannot distinguish between input a1 and a2 but the adversary has presented 2 inputs that has been consistent with the queries but ultimately gives different solutions

Lower bound for decision tree sorting is omega(nlogn), look at tutorial 03!

Running Time of algorithms
→ Consider worst case and best case run times

Example: Checking a Number is Prime or not

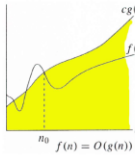
```
IsPrime(k)
{
    For(i = 2 to k − 1) do      k-2 iterations
    {
        If(k%i = 0) return "Not Prime";      1 Instruction
    }
    return "Prime";      The final instruction
}
```

Worst-case time ≈ k
Best-case time= 2

Asymptotic Analysis
Big-oh

Graphical explanation of O-notation

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.



O-notation is an *upper-bound* notation. It makes **no sense** to say $f(n)$ is at least $O(n^2)$.

→ O has a set notation as well

Set definition of O-notation

$$O(g(n)) = \{ f(n) : \text{there exist constants } c > 0, n_0 > 0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \}$$

- O(g(n)) is actually a set of functions.
- Although we write f(n)=O(g(n)), we mean f(n)∈O(g(n))

Big-omega
→ f(n)'s time complexity is at least g(n)
Big-Theta
→ It is both the big-omega and the big-Oh
Little-o and little-omega
→ Little o means f(n) strictly grows less than g(n)
→ Little omega means f(n) strictly grows more than g(n)

Definition 2.2.1 (Big Oh and Big Omega).

- Big Oh (O): We write $f(n) = O(g(n))$, or $f(n) \le O(g(n))$, if there exist constants $c > 0, n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.
- Big Omega (Ω): We write $f(n) = \Omega(g(n))$, or $f(n) \ge \Omega(g(n))$, if there exist constants $c > 0, n_0 > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$.
- Theta (Θ): We write $f(n) = \Theta(g(n))$ if there exist constants $c_1, c_2, n_0 > 0$ such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$. Note that $f(n) = \Theta(g(n))$ iff $f(n) \le O(g(n))$ and $f(n) \ge \Omega(g(n))$ simultaneously.

Definition 2.2.2 (Little Oh and Little Omega).

- Little Oh (o): We write $f(n) = o(g(n))$, or $f(n) < o(g(n))$, if for any $c > 0$, there is an $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0$.
- Little Omega (ω): We write $f(n) = \omega(g(n))$, or $f(n) > \omega(g(n))$, if for any $c > 0$, there is an $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0$.

Let us give a few examples right away to illustrate the definitions:

Using limits to determine time complexity

- Assume f(n), g(n)>0.

- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \Rightarrow f(n) = o(g(n))$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \Rightarrow f(n) = O(g(n))$
- $0 < \lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \Rightarrow f(n) = \Theta(g(n))$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) > 0 \Rightarrow f(n) = \Omega(g(n))$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty \Rightarrow f(n) = \omega(g(n))$

→ Proof exists in slide 40 of lecture 2 on why the limit = 0 means f(n) = o(g(n))
→ big-O is transitive, reflexive, symmetric, complementary

Properties of big-O

- **Transitivity**
  $f(n) = \Theta(g(n)) \,\&\, g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  $f(n) = O(g(n)) \,\&\, g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
  $f(n) = \Omega(g(n)) \,\&\, g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
  $f(n) = o(g(n)) \,\&\, g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$
  $f(n) = \omega(g(n)) \,\&\, g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$
- **Reflexivity**
  $f(n) = \Theta(f(n))$
  $f(n) = O(f(n))$
  $f(n) = \Omega(f(n))$
- **Symmetry**
  $f(n) = \Theta(g(n))$ *iff* $g(n) = \Theta(f(n))$
- **Complementarity**
  $f(n) = O(g(n))$ *iff* $g(n) = \Omega(f(n))$
  $f(n) = o(g(n))$ *iff* $g(n) = \omega(f(n))$

L'Hopital's Rule
→ We can calculate the fraction of the limits using L'Hopital's Rule

$$\lim_{x\to\infty} f(x) = \infty, \lim_{x\to\infty} g(x) = \infty \implies \lim_{x\to\infty}\frac{f(x)}{g(x)} = \lim_{x\to\infty}\frac{f'(x)}{g'(x)}$$

	A	B	O	o	Ω	ω	Θ
(a)	n^3+4n	$(\lg n)^{2022}$	no	no	yes	yes	no
(b)	n^9	1.01^n	yes	yes	no	no	no
(c)	$n^{1.5}$	$n\lg n$	no	no	yes	yes	no
(d)	2^n	3^n	yes	yes	no	no	no
(e)	$\lg(n^4)$	$\lg(n^8)$	yes	no	yes	no	yes
(f)	n^{10}	$n^{\lg n}$	yes	yes	no	no	no

→ The yes and nos are saying that A = O(B) for example (e.g. $n^9 = O(1.01^n)$)
→ $\sqrt{n}$ grows faster than log(n), so logn = o($\sqrt{n}$)

Mathematical Induction

Induction is a method of proof. Typically, induction is applicable whenever we want to prove some statement S_n that depends on some integer parameter n. To perform induction,

1. Establish that the statement is true for the base case.
2. Show that if the statement is true for the n-th case (this is the induction hypothesis), then the statement is also true for the $(n+1)$-th case.
3. Conclude that the statement holds for all cases.

→ Start by trying out the base case, whether the LHS matches the RHS
→ Once it match, make an inductive hypothesis, which is basically assuming what we want to prove is true, then we try to verify if it is true for the n+1 case (e.g. Let P(k) be the statement: .... = ..., assume P(k) is true, show that P(k+1) is true)
→ In verifying if its true, use what we assumed in the inductive hypothesis. Once, proven, remember to give a short conclusion (e.g. Because P(1) is true and P(k) => P(k+1) for all positive integers K, by MI, P(K) holds for all positive integers K)

Example 2. Sometimes we require a more complex inductive hypothesis in order to perform induction. Consider the statement that for all $n \ge 8$, $n = 3a + 5b$ for some non-negative integers a and b.
Here we use a modified induction hypothesis: for some $k \ge 3$, $3k-1$, $3k$ and $3k+1$ can all be written in the desired form.
Base case: For $k = 3$, we have $8 = 3 \cdot 1 + 5 \cdot 1$, $9 = 3 \cdot 3$, $10 = 5 \cdot 2$.
Inductive step: Assume that the induction hypothesis holds for k. For each n, let a_n and b_n be defined such that $n = 3a_n + 5b_n$ if n can be represented in this form.
Then we can write:

$$\begin{aligned} 3k+2 &= (3k-1)+3 \\ &= (3a_{3k-1} + 5b_{3k-1}) + 3 \text{ (applying the induction hypothesis to } (3k-1)) \\ &= 3(a_{3k-1}+1) + 5b_{3k-1} \end{aligned} \quad (1.5)$$

The working is similar for $3k+3$ and $3k+4$ and is left as an exercise. Therefore $3(k+1)-1$, $3(k+1)$ and $3(k+1)+1$ can all be written in the desired form.
Conclusion: Note that all integers $n \ge 8$ can be written as $3k-1$, $3k$ or $3k+1$ for some $k \ge 3$. By induction, for all $n \ge 8$, $n = 3a + 5b$ for some integers $a, b \ge 0$.

Strong Induction
- Assume in the inductive step that everything up to n is true

Proof By Contradiction

Proof by Contradiction

Idea:

1. *Assume* what we want to prove is ***not* true**.
2. Using this, we show that the **consequences is not possible**
   a. Either contradicting what we assumed or
   b. Some other fact we already know to be true
   c. (or both)
3. Thus, **what we *assumed*** must be **incorrect**. So we can conclude that what we want to prove is true

Proof by Contradiction (Example)

To prove: If p^2 is even, then p is even
Proof (by contradiction):

1. Assume that p is actually odd (can be expressed in the form of $2n+1$)
   a. What we know currently is two things:
      i. p^2 is even (what we started off with)
      ii. p is odd (by assumption) and
2. Expand $p^2 = (2n+1)^2 = 4n^2 + 4n + 1$
3. But $p^2 = 4n^2 + 4n + 1$ can be rewritten as $2(2n^2 + 2n) + 1$
4. We deduce that p^2 is odd
5. But this contradicts the fact that p^2 is supposed to be even!

Proving Iterative Algorithms
- We need to prove the invariants of the algorithm, and show that these invariants lead to correctness of the algorithm

1. Initialization: The invariant is true before the first iteration of the loop
2. Maintenance: If the invariant is true before an iteration, it remains true before next iteration
3. Termination: When the algorithm terminates, the invariant provides a useful property for showing correctness

Djikstra Invariant: for every visited node, we have the shortest path from the current node to the visited nodes.

Tower of Hanoi
- Move top n-1 discs from first to second peg, using third as temporary storage. Next, move the biggest disk to currently empty third peg. Finally, move the n-1 disks from the second peg to the third using the first as temp storage

```
Hanoi(n, src, dst, tmp)
1  if n ≥ 1
2       Hanoi(n − 1, src, tmp, dst)
3       Move disk n from src to dst
4       Hanoi(n − 1, tmp, dst, src)
```

Misra-Gries (Track majority in a space-efficient manner from a stream)
→ Like balancing a scale, we track the majority element and its net count more than the minority element. If the count = 0, then the majority_ele = null (the scale is balanced)

```
Misra-Gries(A)
1  id = ⊥
2  count = 0
3  for i = 1 to A.length
4       if A[i] == id
5            count = count + 1
6       elseif id == ⊥
7            id = A[i]
8            count = 1
9       else
10           count = count − 1
11           if count == 0
12                id = ⊥
13 if id == ⊥
14      return "Tie"
15 else return id
```

Euclidean (Find gcd of 2 elements a and b)

```
Euclid(a, b)
1  while b > 0
2       c = a
3       a = b
4       b = c (mod b)
5  return a
```

→ Based on the invariant:

Claim 3.1.4. *If $a \ge 0, b > 0$, then $\gcd(a,b) = \gcd(b, a \bmod b)$.*

Proof. Suppose for some integers $q \ge 0$ and $0 < r < b$, we can write $a = qb + r$, so that $r = a \pmod b$. Let $x = \gcd(a,b)$ and $y = \gcd(b,r)$. We will be done if we can show that x divides y, and also y divides x.
First, let's show that x divides y. Since x divides a as well as b, it must also divide $r = a - qb$. As x is a common divisor of both b and r, it must divide $\gcd(b,r)$.
Similarly, let's show that y divides x. Since y divides b as well as r, it must also divide $a = qb + r$. As y is a common divisor of both a and b, it must divide $\gcd(a,b)$. □

Proving Recursive Problems
- I think using (strong) induction is one of the easier ways to prove recursion? Because recursion has a base case and we just need to identify and prove the inductive hypothesis holds
- Use the T(n) = … to breakdown the complexity of a recursive problem

Binary Exponentiation

Example 5. (Powering) Binary exponentiation is a fast way to find power n of a number A. The idea is we start with base case of $A^0 = 1$. For $n > 0$, if n is odd, $A^n = (A^{(n-1)/2})^2 \cdot A$, and if n is even, $A^n = (A^{n/2})^2$.

```
RECURSE-POWER(A, n)
1  if n = 0 return 1
2  if n is odd return RECURSE-POWER(A, (n − 1)/2)^2 · A
3  else return RECURSE-POWER(A, n/2)^2
```

Karatsuba Algorithm

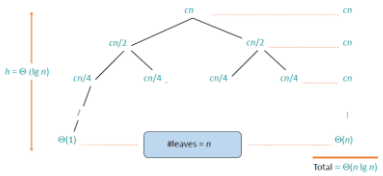
→ Multiply 2 numbers in $O(n^{\log 2(3)})$ instead of $O(n^2)$

Analyzing Time Complexity

1. Recursion Tree

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



→ For recursion tree, try and add up the entire work done by expanding out the entire tree. If u can get the exact number, then that's the tight bound (theta)

→ If you can only make a rough estimation (maybe u know each level is <=n), then u can give a big-oh bound (e.g. O(nlgn))

Master Method

Applies to recurrences of the form:

- $T(n) = a\,T(n/b) + f(n)$,

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).
   - ***Solution:*** $T(n) = \Theta(n^{\log_b a})$.

→ The epsilon value is minus after evaluation logb(a), e.g. logba = n^2, f(n) = n

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.
   - ***Solution:*** $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),
   - ***and*** $f(n)$ satisfies the ***regularity condition*** that $af(n/b) \leq cf(n)$ for some constant $c < 1$.
   - ***Solution:*** $T(n) = \Theta(f(n))$.

The regularity condition guarantees the sum of subproblems is smaller than f(n)

→ To apply master theorem, if the value of the power term in $n^{\log b(a)}$ is different from the one in f(n) then can apply?

Substitution Method

1. Guess the form of the solution
2. Verify by induction

→ Note that for induction, you MUST show that the form that T(n) <= the form you proposed

→ We assume that T(1) = q, where q is some constant

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

- A possible solution to prove that $T(n) = O(n^2)$.
  - i.e. we show that $T(n) \leq c\,n^2$ for $n \geq n_0$.
- Set $c = \max\{2, q\}$ and $n_0 = 1$.
- Base case (n=1): $T(1) = q \leq c(1)^2$.
- Recursive case (n>1):
  - By strong induction, assume $T(k) \leq c\,k^2$ for $n > k \geq 1$.
  - $T(n) = 4\,T(n/2) + n$
  - $\leq 4\,c\,(n/2)^2 + n$
  - $= c\,n^2 + n$
  - $= O(n^2)$. ←This is not correct! You need to show $T(n) \leq c\,n^2$!

→ This proof for $O(n^2)$ is wrong because our ending term is > n^2

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

- [Assume $T(1) = q$ where q is a constant.]
- Correct solution: Show that, for $n \geq n_0$, $T(n) \leq c_1 n^2 - c_2 n$.
- Set $c_1 = q+1$ and $c_2 = 1$ and $n_0 = 1$.
- Base case (n=1): $T(1) = q \leq (q+1)\,(1)^2 - (1)(1)$.
- Recursive case (n>1):
  - By strong induction, assume $T(k) \leq c_1 k^2 - c_2 k$ for $n > k \geq 1$.
  - $T(n) = 4\,T(n/2) + n = 4\,(c_1\,(n/2)^2 - c_2\,(n/2)) + n = c_1 n^2 - 2\,c_2\,n + n = c_1 n^2 - c_2 n + (1 - c_2)\,n$
  - Since $(1 - c_2) = 0$, $T(n) \leq c_1 n^2 - c_2 n$.

→ Tip: Just expand out the equation in terms of c1 and c2 first. Then try and use values of c1 and c2 that can make the expanded equation be less than or equal to the original equation. Then tweak a bit to fit the base case also.

→ $2T(n/2) + n = n\lg n$

→ $4T(n/2) + n = n^2$

Average Case Analysis

→ For algorithms like quicksort, the run time is heavily dependent on the input given, and it might be hard to analyze its runtime, unless we argue in an average case, where we use its probability distribution / expectation to argue

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  // Choose the first element as pivot
2  x = A[p]
3  i = r + 1
4  for j = r downto p + 1
5      if A[j] > x
6          i = i − 1
7          Swap A[i] and A[j]
8  Swap A[i − 1] and A[p]
9  return i − 1
```

→ In e.g. above, Select pivot deterministically to be 1st ele

→ When we analyze average case we say: 'the average-case running time of quicksort with respect to the uniform distribution on permutations of length n is O(nlogn)'

Average-case Analysis of Quick Sort

0	1	2	3	4	5	6	7	8
6	11	42	37	24	5	16	27	2
e_2	e_4	e_6	e_5	e_8	e_3	e_7	e_9	e_1

0	1	2	3	4	5	6	7	8
15	20	49	41	29	4	23	36	3
e_3	e_4	e_6	e_5	e_8	e_2	e_7	e_9	e_1

Let e_i : ith **smallest** element of A.

Observation: The execution of **Quick sort** depends upon the permutation of e_i's and **not** on the values taken by e_i's.

$A(n)$ = Average running time for Quick sort on input of size n
=Expected running time of Quick sort when the input is chosen uniformly at random from the set of all $n!$ Permutations (i.e., expectation is over the random choices of the input).

(average over all possible permutations of $\{e_1, e_2, \ldots, e_n\}$)

Hence, $A(n) = \frac{1}{n!}\sum_{\pi} Q(\pi)$,

where $Q(\pi)$ is the time complexity (or no. of comparisons) when the input is permutation π.

→ Average running time is given by taking the time complexity of each permutation and multiplied by the probability of that permutation

- $G(n, i)$ = average running time of **QuickSort** over $P(i)$.
- So, $G(n, i) = A(i-1) + A(n-i) + (n-1)$
- We have already seen, $A(n) = \frac{1}{n}\sum_{i=1}^{n} G(n, i)$
- So, $A(n) = \frac{1}{n}\sum_{i=1}^{n}(A(i-1) + A(n-i) + n - 1)$

$$= \frac{2}{n}\sum_{i=1}^{n} A(i-1) + n - 1$$

A(i-1) + A(n-i) + n - 1
= A(0) + A(n-1) + n - 1
+ A(1) + A(n-2) + n-1
+...
+ A(n-1) + A(0) + n - 1
All terms appear twice thats why theres a 2/n

Average-case Analysis of Quick Sort

$$A(n) = \frac{2}{n}\sum_{i=1}^{n} A(i-1) + n - 1$$

Now, we are ready to show the asymptotic bound on $A(n)$ using the substitute-and-check method. We claim that $A(n) \leq 2n \ln n$.
The base case $n = 1$ definitely holds. Assume by strong induction that $A(i) \leq 2i \ln i$ for all $i < n$. Then:

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i)$$
$$\leq n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} 2i \ln i$$
$$\leq n - 1 + \frac{2}{n}\int_{i=1}^{n} 2i \ln i$$

The last line uses the fact that for an increasing function f, $\sum_{i=1}^{n-1} f(i) \leq \int_{i=1}^{n} f(i)di$. This is a simple observation about area under the curve. Integrating:

$$A(n) \leq n - 1 + \frac{2}{n}\left(n^2 \ln n - \frac{n^2}{2} + \frac{1}{2}\right) \leq 2n \ln n.$$

→ Quicksort largely preferred over Mergesort irl because it doesn't have recursion stack and has great performance on average, don't really have to worry about the worst cases

→ **We want to make quicksort distribution insensitive. To do so, we can select pivots uniformly at random**

Then, time taken does not depend on initial permutation of A

Randomized Quick Sort

```
QUICKSORT(A, p, r)
  if p < r
    then q ← PARTITION(A, p, r)
         QUICKSORT(A, p, q−1)
         QUICKSORT(A, q+1, r)
```

Pick an element uniformly at random from A and make it the pivot

Assumption: All elements are distinct (if not, break the ties arbitrarily)

Notation : e_i : ith smallest element of array A.

```
QUICKSORT(A, p, r)
  if p < r
    then q ← PARTITION(A, p, r)
         QUICKSORT(A, p,
         QUICKSORT(A, q
```

Maximum over inputs A of length n

Let $T(n)$ be the worst-case number of comparisons.

Note that $T(n)$ is a **random variable.**

→ The number of comparisons is no longer deterministic, so random variable

Randomized Quick Sort Analysis

```
QUICKSORT(A, p, r)
  if p < r
    then q ← PARTITION(A, p, r)
         QUICKSORT(A, p, q−1)
         QUICKSORT(A, q+1, r)
```

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

Let $A(n) = \mathbb{E}[T(n)]$ where the expectation is over the randomness in the algorithm.

Taking expectations and applying linearity of expectations:

$$A(n) = n - 1 + \frac{1}{n}\sum_{q=1}^{n}(A(q-1) + A(n-q)) = n - 1 + \frac{2}{n}\sum_{q=1}^{n-1} A(q)$$

This is the same as the recurrence for average-case quicksort!

→ For random variables, we use expectation to analyze. This shows that A(n) = O(nlogn). While the recurrence looks similar to the average case one, in average case, each run time is deterministic, but for this case, we are analyzing the expected runtime for any case. Because now each case is non-deterministic

Geometric Distribution

- Suppose you flip a fair coin until it comes up heads. What is the expected number of times you need to flip?

- Let X be the number of times. Note that X is a random variable.
- X follows a **geometric distribution** with probability p = 1/2
- Pr[X = 1] = 1/2, Pr[X = 2] = 1/4, Pr[X = 3] = 1/8, ...
- **Fact:** E[X] = 1/p

→ If probability of an event is 1/5, then expected number of times before event happens is 1/p = 5

Las Vegas vs Monte Carlo

→ Las Vegas – Algorithm always correct, running time is a random variable

→ Monte Carlo – Algorithm not always correct, running time is fixed

Coupon Collector Problem

→ Probability of new coupon after i-1 coupon is collected: Pi = (n-(i-1))/n

Let T_i be the number of draws used to collect the i-th coupon and T be the total number of draws.

$$T = \sum_{i=1}^{n} T_i \qquad \text{Total number of draws}$$

$$E[T] = E\left[\sum_{i=1}^{n} T_i\right] \qquad \text{Expected value}$$

$$= \sum_{i=1}^{n} E[T_i] \qquad \text{Linearity of expectation}$$

→ Add the expected time to collect each coupon, which is 1/Pi = n/(n-(n-i))

Harmonic Series:

$$E[T] = \sum_{i=1}^{n} E[T_i]$$
$$= \sum_{i=1}^{n} \frac{n}{n-(i-1)}$$
$$= \frac{n}{n} + \frac{n}{n-1} + \cdots + \frac{n}{1}$$
$$= n \cdot \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}\right)$$
$$= n \cdot H_n$$
$$= \Theta(n \lg n)$$

where H_n is the n-th harmonic number and is $\Theta(\lg n)$

Hashing

Dictionary: supports inserts, search, delete

Desired Properties of dictionary/table

→ Minimize collision, querying and deleting time should take Θ(h(x)), which should ideally be O(1). Worst case when all inserted keys hash to same location

→ Minimize storage space, best is M = O(N), where M is number of buckets

→ h should be easy to compute and hopefully in constant time

→ Collision means that adversary can find N elements that hash to same value. M-1 holes with N-1 values, 1 hole with N values because of the + 1 term below

Adversary strikes back!

- If U is large, then for any hash function with small M, there are many keys which all hash to the same location.
- **Claim:** If $|U| \geq (N-1)M + 1$, for any $h: U \to [M]$, there is a set of N elements having the same hash value. Here and later, $[M]$ denotes the set $\{1, 2, \ldots, M\}$.
- **Proof: Pigeonhole principle.** If every slot in the hash table had $< N$ elements from U mapping to it, then $|U| \leq (N-1)M$. Contradiction!

→ Randomization, choose hash function randomly. **Note:** hash function itself is not random. We only randomize at the start before choosing hash function

Universal Hashing

Definition: Suppose $\mathcal{H}$ is a set of hash functions mapping U to $[M]$. We say $\mathcal{H}$ is ***universal*** if for all $x \neq y$:

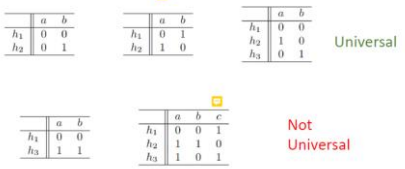
$$\frac{|h \in \mathcal{H} : h(x) = h(y)|}{|\mathcal{H}|} \leq \frac{1}{M}.$$

(# of hash functions for which x and y collide)

For any $x \neq y$, if h is chosen uniformly at random from a universal $\mathcal{H}$, there's at most $\frac{1}{M}$ probability that $h(x) = h(y)$.

→ A set of hash functions is universal if the probability of **collision** of a certain value is 1/numBuckets. Basically we calculate the hash value of any 2 values in the universe using every single hash function available in our family. Then we calculate how many of them had collision. For all possible values, if the probability is <= 1/M, then it is a universal family

Universal Hashing Examples



Expected Costs of N insertions / deletions / queries: O(N)

Claim: Suppose $\mathcal{H}$ is a *universal* family of hash functions mapping U to $[M]$. For any sequence of N insertions, deletions and queries, if $M \geq N$, then the expected total cost for a random $h \in \mathcal{H}$ is $O(N)$.

PROOF
- Each operation costs $O(1)$ time in expectation by previous claim.
- By linearity of expectations, total cost is $O(N)$.

Expected Number of collisions of any element with Xn

Properties of universal hashing. The following captures one of the main implications of universality.

Claim 5.3.2. *Suppose the hash function is drawn from a universal family $\mathcal{H}$ as above. Then, for any N elements $x_1, \ldots, x_N \in \mathcal{U}$, the expected number of collisions between x_N and the other elements is $< N/M$.*

Proof. From Definition 5.3.1, each element $x_1, \ldots, x_{N-1} \in \mathcal{U}$ has at most $1/M$ probability of collision with x_N (over a random choice of h). To calculate the expected number of collisions, we use the *indicator random variable method* that is **extremely** useful for such problems. (Please see the supplementary review of indicator random variables for much more on this method.)

For $i < N$, let C_i be the random variable that is 1 if x_N collides with x_i and 0 otherwise. $\mathbb{E}[C_i] = 1 \cdot Pr[C_i = 1] + 0 \cdot Pr[C_i = 0] = Pr[C_i = 1] \leq \frac{1}{M}$. By linearity of expectation, the expected number of collisions between x_N and the other elements is

$$\mathbb{E}\left[\sum_{i=1}^{N-1} C_i\right] = \sum_{i=1}^{N-1} \mathbb{E}[C_i] \leq (N-1) \cdot \frac{1}{M} < \frac{N}{M}.$$

→ Proof is using indicator random variable, let Ci be the random variable that is 1 if Xn collides with Xi and 0 otherwise, then use linearity of expectations to sum up the probability of collision for all Ci, which is N/M.
→ **Impt:** Take note that this is the probability of collision between Xn and other elements

Expected number of pairs (I,j) where h(xi)=h(xj)

→ This is counting collision among any 2 arbitrary pairs
→ Condition for this to hold true is that M > N

Claim 5.3.4 (Total number of collisions). *Suppose a hash function h is drawn from a universal family $\mathcal{H}$ as above. If $x_1, \ldots, x_N$ are added to the hash table, and $M > N$, the expected number of pairs (i,j) such that $h(x_i) = h(x_j)$ is $< 2N$.*

Proof. Let A_{ij} equal 1 if $h(x_i) = h(x_j)$ and 0 otherwise. Then:

$$\mathbb{E}\left[\sum_{1 \leq i,j \leq N} A_{ij}\right] = \sum_{1 \leq i,j \leq N} \mathbb{E}[A_{ij}]$$
$$= \sum_{i=1}^{N} \mathbb{E}[A_{ii}] + \sum_{i \neq j} \mathbb{E}[A_{ij}]$$
$$\leq N \cdot 1 + N(N-1) \cdot \frac{1}{M} < 2N$$

→ Idea: Each element collides with itself. Theres N of those elements
→ For the other elements, estimate that the number of times each element collide with others is N/M (from prev claim). There are N-1 possible pairs, so + N(N-1)/M. Since M > N then this value is definitely < N so total <2N

Universal Family of Hash Functions

→ If the universe is u-bit strings, and the number of buckets M = 2^m then we can construct the set of matrices with m rows and each having u elements (u columns) that is a universal family of hash functions

- Suppose U is indexed by u-bit strings, and $M = 2^m$. For any binary matrix A with m rows and u columns:

$$h_A(x) = Ax \pmod 2$$

Claim: $\{h_A : A \in \{0,1\}^{m \times u}\}$ is universal.

→ A is m x u, value x is u x 1, final hash is m x 1. Size of family is 2^{mu}

Correctness

→ We want to find out with different x and y values, what is the probability that Ax = Ay
→ Since x and y are different values z = x – y != 0
→ But Az = 0.
→ The idea is arguing is that since the probability of any combination of values that make up the matrix is uniformly distributed, Pr[Az=0] = 1/M.
→ Imagine trivial case where everything in z is 0 except the ith-value. Means Az is basically the value of the ith column of A. (Remember z is a column vector and A is a matrix). Remember A is also uniformly random.
With all these info, whats the probability that Az = 0? If only the ith value of z is 1, then Az is basically the sum of the ith column after you do matrix multiplication. Then cus theres M possible values from mod M (0 – M-1) so the probability its 0 is 1/M
→ This holds for every column, and since each column is independently created, overall, the probability is still 1/M. Intuition for this is the coin flip example in lecture

- Warm-up for general case: If you flip a fair coin independently k times, what is the probability that the number of times it comes up heads is even?
- General case: Suppose … ure notes or presentation.

songy: no matter what the value of the first k-1 coins are, for the last one, theres a 50% chance to make it even, so probability is 1/2

Universal Hash Summary

→ Space overhead to specify a particular hash function is O(log|U|.log|M|) or O(um). U is large but logU is reasonable (e.g. 256 bits for universe of 2^{256}
→ Time complexity given by matrix multiplication, which is quite expensive

Perfect Hashing

→ Construct a data structure where query(y) (y is any element from U) is O(1) worst case
→ Condition: We know the X1 to Xn elements to be hashed
→ There can be such a hash from the universal family of hash functions H, if numBuckets M = N^2! This is similar to the proof on calculating the number of collisions between any 2 elements from X1 to Xn.

Claim: If $\mathcal{H}$ is *universal* and $M = N^2$, then if h is sampled uniformly from $\mathcal{H}$, the expected number of collisions is < 1.

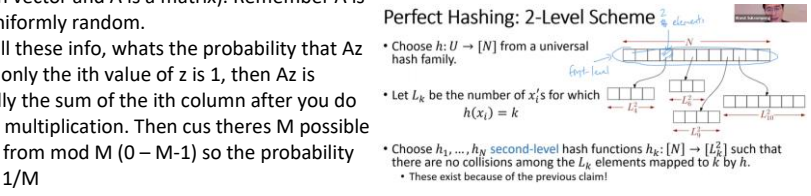
PROOF
- For $i \neq j$, let A_{ij} equal 1 if $h(x_i) = h(x_j)$, and 0 otherwise.
- By universality, $\mathbb{E}[A_{ij}] = \Pr[A_{ij} = 1] \leq 1/N^2$.
- $\mathbb{E}[\#\text{collisions}] = \sum_{i \neq j} \mathbb{E}[A_{ij}] \leq \binom{N}{2} \frac{1}{N^2} < 1$.

→ Since expectation less than 1, there must be some hash where collision = 0, because if all of the hash functions have at least 1 collision, the expectation will be at least 1

2 Level Scheme Perfect Hashing

→ Hash once, then hash again.
→ Based on the number of elements in each of the buckets for the first hash, we create another level where the number of second-layer buckets is squared of the number of elements in each first-layer bucket, because there exists a hash function with no collision

Perfect Hashing: 2-Level Scheme



→ Question is asking what is the total size of the second level buckets (We are hoping that this is O(N) not $O(N^2)$)
→ Basically, Lk^2 = the squared of the number of values that hash to the same bucket, and the sum of Lk^2 = Aij because Aij is basically counting the number of (I,j) pairs. And we established previously its <2N so space is O(N)

Perfect Hashing: 2-Level Scheme

Claim: If $\mathcal{H}$ is *universal*, then if h is sampled uniformly from $\mathcal{H}$:

$$\mathbb{E}\left[\sum_k L_k^2\right] < 2N.$$

- For $1 \leq i, j \leq N$, define $A_{ij} = 1$ if $h(x_i) = h(x_j)$ and $A_{ij} = 0$ otherwise
- **Crucial observation:**

$$\sum_k L_k^2 = \sum_{i,j} A_{ij}$$

- $\mathbb{E}[\sum_{i,j} A_{ij}] = \sum_i \mathbb{E}[A_{ii}] + \sum_{i \neq j} \mathbb{E}[A_{ij}] \leq N \cdot 1 + N(N-1) \cdot \frac{1}{N} < 2N$

Pairwise Independence

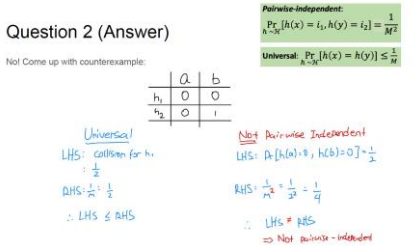
A **pairwise independent** family $\mathcal{H}$ of hash functions mapping $\mathcal{U}$ to $\{1, \ldots, M\}$ has the property that for any two distinct universe elements x, y and for any two hash values i_1, i_2:

$$\Pr_{h \sim \mathcal{H}}[h(x) = i_1, h(y) = i_2] \leq \frac{1}{M^2}.$$

→ The <= equality should actually be equal, because for it to be pairwise independent the probability of any value x hashing to some location h(x) must be 1/M (If it is less than that then there will be another h(y) that is >1/M, which will contradict the pairwise independence definition

Pairwise family is universal, universal family does not mean pairwise independent

Question 2 (Answer)



- Intuitively:
  - In Pairwise Independent Family, you can ***freely*** choose any pair of hash values i_1 and i_2
  - In Universal Family, you are ***limited*** to hashes of x and y that collide (equal i_1 and i_2)

Pairwise-Independent: $\Pr_{h \sim \mathcal{H}}[h(x) = i_1, h(y) = i_2] = \frac{1}{M^2}$
Universal: $\Pr_{h \sim \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{M}$

Hash Table Resizing

→ Hash tables should be M = O(N) for the costs of insertions/search/deletion, but we might not know how many N we have, so its hard to set M optimally.
→ Solution? We rehash when N is too large and choose a new hash function. Rehashing is costly but happens infrequently, we can use amortized cost to analyze such an operation

Fingerprinting

String pattern matching

Problem: We want to see if a pattern string P occurs as a substring of a text string T

Naïve Solution:

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4      if P[1..m] == T[s + 1..s + m]
5          print "Pattern occurs with shift" s
```

- Time complexity is $\Theta((n-m+1)m)$. If $m = n/3$, then this is $\Theta(n^2)$.
- Can we do better if m is $\Omega(n)$? What if we use randomization?

→ (n-m+1) is the number of iterations through the string T. Each check at line 4 takes Θ(m) time.
→ If m is length like n/3, then it is basically a $\Theta(n^2)$ algo

How to compare strings faster?

→Using hashing. We design some h that outputs a number, and we can check is h(x) == h(p) in constant time
→ Costs incurred will be in hashing x and p.

- Runtime for equality check: $|\text{hash}_P| + |\text{hash}_X| + O(1)$
- Total runtime for pattern matching: $|\text{hash}_P| + (n-m+1)(|\text{hash}_X| + O(1))$ (Cost of hashing P)

→ If we want to hash the strings, it takes O(m) time, so this algo has no improvement yet

Rolling Hash

→ What if we can design a hash function h that can update h(T[1…m]) to h(T[2…m+1]) in constant time? More generally: h(T[s+1…s+m]) to h(T[s+2…s+m+1]) for all s
Runtime:

- Total runtime for pattern matching: $|\text{hash}_{T[1\ldots m]}| + |\text{hash}_P| + (n-m+1)(O(1)+O(1)) = O(m+n)$

Time to hash T[1..m] and hash(p) initially. After that, all checks will take O(1) time, so m + m + n – m + 1 = O(m + n)

Division Hash

→ Hash a number in O(b) time, where b is the number of bits the number takes

- Choose p to be a random prime number in the range $\{1, \ldots, K\}$.
  - Fact: # primes in range $\{1, \ldots, K\}$ is $> K/\ln K$
- Define, for any integer x:

$$h_p(x) = x \bmod p$$

- If p is small and x is b-bits long in binary, hashing takes $O(b)$ time.
- Will show that hash family $\{h_p\}$ is "approximately" universal.

→ There are K/lnK primes from 1 to K
→ With an appropriate K chosen, the hash family comprising of all possible primes within 1 to K is approximately universal

Probability of Collision for Division Hash

Probability of Collision

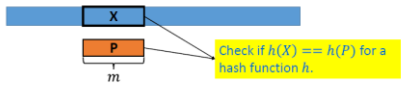
Claim: If $0 \le x < y < 2^b$, then:

$$\Pr_p[h_p(x) = h_p(y)] < \frac{b \ln K}{K}.$$

Proof:
- $h_p(x) = h_p(y)$ when $y - x = 0 \pmod p$. Let $z = y - x$.
- Note $z < 2^b$, so z can have at most b distinct prime factors. (Why?)
- p divides z if p is one of these $\le b$ prime factors.
- Prob of this happening is $< b/\left(\frac{K}{\ln K}\right)$.

→ Probability is blnK/K, compared to 1/K?
Example:

Equality Check Analysis: Example



- Suppose m is 1 million. Naïve equality check would take $\Omega(m)$ time.
- Set K to be 100 million
- If $X \neq P$, $\Pr_p[h(X) = h(P)] < 10^6 \cdot \frac{19}{10^8} < \frac{1}{5}$.
- Note that $K < 2^{32}$, so $h(X)$ and $h(P)$ can be stored in one machine word. They can be compared in constant time!

→ If we set a K = 200mnln(200mn), the probability of collision (false positive) < 1%

Claim: Recall T and P are n and m bits long, respectively. Let $K = 200mn \ln(200mn)$. Then, probability of getting a false positive is < 1%.

Proof:
- Let E_i be the event that $T[i \dots i+m]$ and P are unequal but have the same hash.
- $\Pr[E_i] < m \frac{\ln(200mn \ln(200\,mn))}{200mn \ln(200\,mn)} < \frac{1}{200n}\left(1 + \frac{\ln(\ln(200\,mn))}{\ln(200\,mn)}\right) < \frac{1}{100n}$.
- $\Pr[\cup_i E_i] \le \sum_i \Pr[E_i] < \frac{1}{100}$. (First inequality uses "union bound")

→ Remember P(collision) = blnK/K. In this case, P is m bits long so b = m. So if the probability of each check colliding is 1/100n, as we slide our substring of T across, adding all of them up together (we do this n-m+1) times, the Pr(collision) is less than 1%!

Complexity analysis of prime <= K in division hash
→ If K = 200mnlg(200mn) any prime K can be stored as a bit string of length lgK = O(lgn)
→ We can also compare equality of hashes in constant time (as mentioned for division hash)

How can we roll division hash?
→ Represent X and X' in the following form
- Let X and X' be the binary numbers corresponding to $T[1 \dots m]$ and $T[2 \dots m+1]$.

$$X = \sum_{i=1}^{m} T[i] \cdot 2^{m-i} \qquad X' = \sum_{i=1}^{m} T[i+1] \cdot 2^{m-i}$$

- E.g., if $T = [a_1, a_2, a_3]$ with $m = 2$, then $X = 2a_1 + a_2$ and $X' = 2a_2 + a_3$. You can "roll" from X to X' by $X' = 2X - 4a_1 + a_3$.

→ How to roll depends on the length of P (string to compare). Generalized:

- $X' = 2X - 2^m T[1] + T[m+1]$.

→ Now how do we roll $h_p(X)$ to $h_p(X')$?
- Division hash is **linear**:

$$h_p(X') = 2h_p(X) - T[1] \cdot h_p(2^m) + T[m+1] \pmod p$$

- Given $h_p(X)$ and $h_p(2^m)$, can get $h_p(X')$ in constant time!

→ Recall: (a+b)modp = (amodp + bmodp)modp
(ab)modp = [(amodp)(bmodp)](modp)
- Monte Carlo algorithm with error probability $< 1\%$ and runtime $O(m+n)$.
- Recap:
  1. Pick random prime p in range $\{1, \dots, [200mn \ln(200\,mn)]\}$
  2. Compute $h_p(P)$, $h_p(2^m)$ and $h_p(T[1 \dots m])$
  3. Check if $h_p(P) == h_p(T[1 \dots m])$
  4. For each $i = 1 \dots n - m$
     i. Update $h_p(T[i \dots i+m-1])$ to $h_p(T[i+1 \dots i+m])$, using $T[i]$, $T[i+m]$, and pre-computed $h_p(2^m)$
     ii. Check if $h_p(T[i+1 \dots i+m])$ equals $h_p(P)$

Streaming
→ We want frequency of an item by the end of a stream, fi.
→ Denote sum of all frequencies to be M
- At the end of the stream, **frequency** f_i of an item i is its net count.
  - In above example, $f_3 = 0, f_1 = 2, f_7 = 1$.
  - We assume that $f_i \ge 0$ for all i.
- Let M denote the sum of all frequencies (i.e., total net count) at the end of the stream.
  - Above, $M = 0 + 2 + 1 = 3$.

→ **Frequency Estimation:** We say a stream provides a epsilon-estimate if:
- Call an approximation $\hat{f}_i$ to be **ϵ-approximate** if:

$$f_i - \epsilon M \le \hat{f}_i \le f_i + \epsilon M$$

→ We use a hash table with some hash function to estimate fi

- Choose a hash function $h: \mathcal{U} \to \{1, \dots, k\}$
- Initiate an empty table A of size k
- If j is added, increment $A[h(j)]$
- If j is deleted, decrement $A[h(j)]$
- At end of stream, to estimate f_i, return $A[h(i)]$

Analysis of using hash table for approximation fi
→ A[h(i)] >= fi. It will never be an underestimate
→ A[h(i)] is the sum of all fj that hash to the location of h(i).

$$A[h(i)] = \sum_{j:h(j)=h(i)} f_j$$

- If h is drawn from a universal family, $\mathbb{E}[A[h(i)] - f_i] \le M/k$.
  - See analysis in lecture notes or in presentation.
- Output is in expectation an ϵ-approximation if $k = 1/\epsilon$. Space is $O\left(\frac{1}{\epsilon}\right)$ counters plus cost of storing hash function, $O(\log U \cdot \log(k))$.

→ Expected number of collision with some element is N/k (Refer to image above analysis header, our universe hashes to k buckets) when we input hash N elements
→ So M is counting all the fj that does not include fi (they collided with fi)
→ If k = 1/epsilon, then E[A[h(i)] – fi] <= M/(1/epsilon) = M*epsilon. So our estimations Will be epsilon-approximate since they exceed by an expected amount of M*epsilon
→ Space is O(1/epsilon) or O(k) to store the hashmap, because our hashmap has k buckets.
→ Space to store hash function is O(logU.logk), assuming we are using the matrix universal hash function, where the matrix size is logU * logk

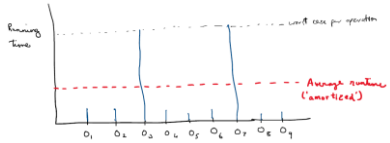
Amortized Analysis

Asymptotic Notation for Multiple Parameters
- What does $O(m+n)$ or $O(mn)$ mean?
- For two functions $f(m,n)$ and $g(m,n)$, we say that $f(m,n) = O(g(m,n))$ if there exist constants c, m_0, n_0 such that

$$0 \le f(m,n) \le c \cdot g(m,n)$$

for all $m \ge m_0$ or $n \ge n_0$.

Amortized Analysis
- Motivating example: Suppose there is a sequence of n operations, o1,o2…on
- Let t(i) be the time complexity of i-th operation o_i, where it is the worst case time complexity of any of the n operations.
→ We would calculate the n operations to have time complexity of nf(n), which is grossly wrong is the other operations all took O(1) time



→ Average (amortize) the run time across all operations despite having a few expensive ones

Binary Counter
- Objective: Count total bit flips (0→1, 1→0) during n increments.
T(n) = total no. of bit flips during n increments
Aim: We want to get a tight bound on T(n)

Ctr	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0
9	0	1	0	0	1
10	0	1	0	1	0
11	0	1	0	1	1

Objective: Count total Bit flips (0 → 1, 1 → 0) during n increments
$T(n)$ = total no. of bit flips during n increments
Aim: To get a tight bound on $T(n)$

```
INCREMENT(A)
1. i ← 0
2. while i < length[A] and A[i] = 1 do
3.     A[i] ← 0          ▷ flip 1→0
4.     i ← i + 1
5. if i < length[A]
6.     then A[i] ← 1     ▷ flip 0→1
```

→ Flip 1 to 0 as long as A[i] = 1
→ Flip last bit (which is currently a 0) to 1

Count Bit Flips: Attempt 1

Ctr	A[4]	A[3]	A[2]	A[1]	A[0]	Cost
0	0	0	0	0	0	0
1	0	0	0	0	1	1
2	0	0	0	1	0	2
3	0	0	0	1	1	1
4	0	0	1	0	0	3
5	0	0	1	0	1	1
6	0	0	1	1	0	2
7	0	0	1	1	1	1
8	0	1	0	0	0	4
9	0	1	0	0	1	1
10	0	1	0	1	0	2
11	0	1	0	1	1	1

Objective: Count total Bit flips (0 → 1, 1 → 0) during n increments
$T(n)$ = total no. of bit flips during n increments
Aim: To get a tight bound on $T(n)$
Attempt 1:
Let $t(i)$ = no. of bit flips during ith increment

$$T(n) = \sum_{i=1}^{n} t(i)$$

In the worst case, $t(i) = k$
→ $T(n) = O(nk)$
Is this a tight bound?

→ Since this is a k-bit binary counter, at worst, we flip k bits in one operation, giving us time complexity of T(n) = O(nk) for n operations.

Ctr	A[4]	A[3]	A[2]	A[1]	A[0]	Cost	Total Cost
0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1
2	0	0	0	1	0	2	3
3	0	0	0	1	1	1	4
4	0	0	1	0	0	3	7
5	0	0	1	0	1	1	8
6	0	0	1	1	0	2	10
7	0	0	1	1	1	1	11
8	0	1	0	0	0	4	15
9	0	1	0	0	1	1	16
10	0	1	0	1	0	2	18
11	0	1	0	1	1	1	19

Objective: Count total Bit flips (0 → 1, 1 → 0) during n increments
$T(n)$ = total no. of bit flips during n increments
Aim: To get a tight bound on $T(n)$
Attempt 2:
Let $f(i)$ = no. of times ith bit flips

$$T(n) = \sum_{i=0}^{k-1} f(i)$$

$f(0) = n$
$f(1) = n/2$
$f(2) = n/4$
$f(i) = n/2^i$

Much better than $O(nk)$ since $k \approx \log n$

→ $T(n) = n \sum_{i=0}^{k-1} 2^{-i} < 2n$

$n\left(1 + \frac{1}{2} + \frac{1}{4} + \dots\right)$

→ Summing up the total cost, we see that the number of times the 0th bit flip, f(0) = n, f(1) = n/2, f(2) = n/4…
→ Total cost T(n):

→ $T(n) = n \sum_{i=0}^{k-1} 2^{-i} < 2n$

$n\left(1 + \frac{1}{2} + \frac{1}{4} + \dots\right)$

→ Much better than O(nk)
→ Amortized cost per increment = T(n) / n < 2n/n = 2 = O(1)
→ Amortized analysis guarantees the average performance of each operation in the worst case.

Another Example: Queues
→ Consider a queue with 2 operations:
Insert(x): Insert element x into queue
Empty(): Empty queue, implemented by deleting all elements one by one
→ What is the worst case running time for a sequence of n operations?
- Cost of a single insert: O(1)
- Cost of single empty: O(n) (at most n elements inserted)

Amortized Analysis: Queues
- Empty is a sequence of deletes where each delete removes one element from the front of the queue
- #DELETEs <= #INSERTs
- If there are k inserts in the sequence, sum of cost of all the empty's is <= k
- Total cost: <= k + k = 2k <= 2n. Amortized cost is O(1)

Types of Amortized Analysis
- Aggregate Method, Accounting Method, Potential Method
- Aggregate method is simple but does not allow a specific amortized cost to be allocated to each operation.

Accounting Method (Banker's Method)
- Charge ith operation a fictitious amortized cost c(i)
- Idea: Amortized cost c(i) is a fixed cost for each operation, while true cost t(i) varies depending on when the operation is called.
- For all n, amortized cost must satisfy

$$\sum_{i=1}^{n} t(i) \le \sum_{i=1}^{n} c(i) \text{ for all } n$$

→ Sum of true cost must be <= amortized cost.
→ Total amortized cost provides upper bound on total true cost.
→ Using banker's method, the amount of money in the bank must never go below 0
→ The fixed amortized cost c(i) will typically be more than the true cost t(i)
→ Extra amount we pay for cheap operations early on will pay for expensive operations in advance
→ There must be always enough credit to pay for true cost in analysis
→ Different operations can have different amortized costs

Amortized Analysis of Queues (Banker's Method)
→ For insert, set amortized cost to 2 (true cost is 1)
→ For empty, set amortized cost to 0 (True cost is size of queue)
→ Whenever an element is inserted, we pay an extra 1. This extra 1 can be used to pay for delete later.
→ Total cost is at most 2 * #INSERT <= 2n.

Banker's Method for Binary Increment
- In binary increment, we notice that every bit that was flipped to 0 was once flipped from 1. So, we can charge the flipping of 0 to 1 $2, of which any flips from 1 to 0 can then cost $0.

Accounting Analysis of Binary Increment

- Charge $2 for each 0 → 1
  $1 pays for the actual bit setting.
  $1 is stored in the bank.
  At some point of time, this bit must have been set
- **Observation:** At any point, every 1 bit in the counter has $1 on its bank.
- Use that $1 as credit to pay for resetting 1 → 0. (reset is "*free*")

Example:

0	0	0	$1^{\$1}$	0	$1^{\$1}$	0
0	0	0	$1^{\$1}$	0	$1^{\$1}$	$1^{\$1}$ Amortized Cost = $2
0	0	0	$1^{\$1}$	$1^{\$1}$	0	0 Amortized Cost = $2

→ Since bank balance never drops below 0, the sum of the amortized costs provides an upper bound on the sum of the true costs.
Claim: After i increments, the amount of money in the bank is the number of 1's in the binary representation of i.

- **Claim**: After i increments, the amount of money in the bank is the number of 1's in the binary representation of i.
- **Proof**:
  - Every time we set a bit 0 → 1, we pay $2.
  - $1 is used to flip the bit while $1 is stored in bank.
  - Every time we reset a bit from 1 → 0, we use $1 from bank to flip the bit.
  - Hence, the amount of money in the bank is the number of 1's in the binary representation of i.

→ Since the number of 1's in the binary representation of I is non-negative, the bank balance is always non-negative

- Conclusion:
  - Amortized cost for each increment = $2 = O(1)$
  - Amortized cost for n increments = $2n = O(n)$
  - Actual cost for n increments = $O(n)$

Actual cost ≤ Amortized cost

Potential Method

Φ: Potential function associated with the algorithm/data-structure (as opposed to with specific object in data-structure)
Φ(i): Potential at the end of ith operation
Ci: Amortized cost of ith operation
ti: True cost of ith operation
→ Must show that potential = 0 at the start

Key relation:

Ci = ti + Φ(i) – Φ(i-1)

Amortized cost of the operation is the true cost of the operation + the difference in potential (which we can think of as taking out extra money from the bank.)

Adding up the above relation for $i = 1, 2, \ldots, n$:

$$c_1 + c_2 + \cdots + c_n \geq t_1 + t_2 + \cdots + t_n + \phi(n) - \phi(0)$$

→ Typically, Φ(0) = 0 (we start our bank with 0 dollars). So as long as Φ(n) >= 0 for all n, the amortized cost is an upper bound of the true cost, i.e.

$$c_1 + c_2 + \cdots + c_n \geq t_1 + t_2 + \cdots + t_n$$

Potential Method on Queue

Potential Function: Number of elements in the queue after ith operation
We have Φ(0) = 0 Moreover, Φ(n) >= 0 for all n.
Amortized cost for INSERT:

$$c_i = \underbrace{t_i}_{1} + \underbrace{\phi(i) - \phi(i-1)}_{1} = 1 + 1 = 2$$

Amortized cost for EMPTY (suppose there are k elements in the queue):

$$c_i = t_i + \phi(i) - \phi(i-1) = k + 0 - k = 0$$

→ For binary increment, potential function is the number of 1-bits.

Dynamic Table (Potential Method)

Goal: Make the table as small as possible, but large enough so it wont overflow. Expand the table when it is full, transferring all items from old table to new table, which will be twice the size

Notations

Some Notations

- n: number of elements in the table.
- createTable(k): A system-call that creates a table of size k and returns its pointer.
- size(T): the size of table T.
- copy(T,T'): copies the contents of table T into T'.
- free(T): free the space (return the space to OS) occupied by table T.

Algorithm for Insertion

```
If (n = 0)
    T ← createTable(1);
Else
    If(n = size(T))    // Table is full
    {   T' ← createTable( 2n);
        copy(T,T');
        free(T);
        T← T'
    }
Insert x into T;
n← n + 1;
```

→ If we do worst-case analysis and say that each insert takes O(n) time, then the cost of n insertions is n.O(n) = O(n²). But this is not tight.
→ Observe that once the table is full, we create a table of double the size, it will take O(1) time for the next insertions until overflow happens again, i.e. the heavy operation occurs when n-1 is a power of 2 (e.g. 5th operation need to create new table)

Aggregate Method

Aggregate Method

Cost of n insertions = $\sum_{i=1}^{n} t(i)$
$\leq n + \sum_{j=0}^{\lfloor \log(n-1) \rfloor} 2^j$ (Geometric series) $= 2^{\lfloor \log(n-1) \rfloor + 1} - 1 \leq 2(n-1)$
$\leq 3n$

Thus, the average cost of each insertion in dynamic table is $= O(n)/n = O(1)$.

→ Add all the costs incurred and divide over n operations

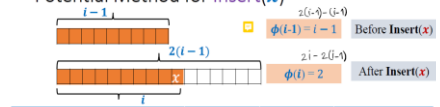
Accounting Method

→ We pay $3 for insertions, where $1 is used to insert into the current table, $1 for moving the element when the table expands, $1 for moving an element that was moved before.

Potential Method

→ Potential is the bank itself, how do we count how much money we have in the bank? For example if we currently have 3 elements in a table of size 4, the potential is 2 (for that 1 element that was added only when the table was size 4). So the potential function is (2*3 elements) – (2*tablesize/2) = 2i – size(T).

Potential Method for Insert(x)



Operation Insert(x)	Actual Cost	$\Delta\phi_i$	Amortized Cost
Case 1: when table is not full	1		
Case 2: when table is already full	i	$3 - i$	3

$\phi(i) = 2i - size(T)$

Potential Method for Insert(x)

Operation Insert(x)	Actual Cost	$\Delta\phi_i$	Amortized Cost
Case 1: when table is not full	1	2	3
Case 2: when table is already full	i	$3 - i$	3

Amortized cost of n insertions = $3n = O(n)$
Actual cost of n insertions = $O(n)$

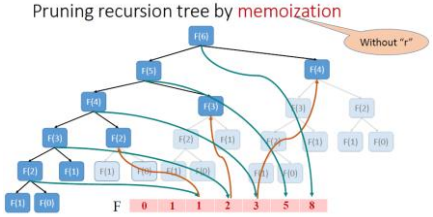
Summary

→ Can use aggregate, accounting, potential methods. Potential is usually the hardest to define at first but very easy to complete the question after. Accounting might be hard to explain how to delegate the cost for each operation.
→ Accounting Method: Show the individual costs for each operation + money in bank will never go below 0
→ Potential Method: Show the amortized costs of each operation + that the potential never go below 0

Dynamic Programming

- Top down vs bottom up

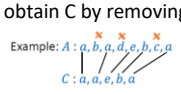
Pruning recursion tree by memoization



→ Top down – "memoization"
→ Bottom Up – "build from the base cases"

Longest Common Subsequence

- C is said to be a subsequence of A if we can obtain C by removing zero or more elements of A



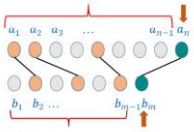
A more formal definition:
C is a subsequence of A if there exists k integers: $1 \leq i_1 < \cdots < i_k \leq n$ s.t. for all $1 \leq j \leq k$ $C[j] = A[i_j]$

→ For brute force, we can permutate all possible 2^n subsequences of A, then for each subsequence, we check whether there is an existing subsequence of B in O(m) time. So, the brute force will take $O(m2^n)$. Is there a faster way?

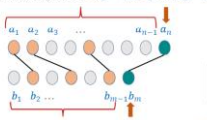
DP: Longest Common Subsequence

→ We start looking at the string from the back. For lcs(m,n), If 2 letters match, then len = 1 + lcs(m-1,n-1), and if they don't match, len = max{lcs(m-1,n) + lcs(m,n-1)}.
→ Why is it that if the 2 letters match, then len = 1 + lcs(m-1,n-1) is true?
→ We prove by contradiction. The idea of the proof is that if we assume that the optimal solution does not contain the last letter a_m, means the answer to S is actually a subsequence of (a1,..,a_{m-1},b1,…,b_{n-1}), and if that is the case, we can just add the last matching letter to get a longer subsequence

How does LCS(n,m) look like when $a_n = b_m$?



Proof: (Proof by contradiction)
- If the last symbol in S =LCS(n,m) is not the same as $a_n (= b_m)$, then that last symbol must be part of $a_1, \cdots, a_{n-1}$ and $b_1, \cdots, b_{m-1}$.
- So, S is actually a subsequence of $a_1, \cdots, a_{n-1}$ and $b_1, \cdots, b_{m-1}$.



This type of argument is also referred to as cut-and-paste argument

Proof: (Proof by contradiction)
- Now we can append a_n with S (i.e., $S :: a_n$) and get a subsequence of length one more
- Thus S cannot be the largest subsequence of $a_1, \cdots, a_n$ and $b_1, \cdots, b_m$ (Contradiction)

→ If the 2 letters don't match, then the longest length is the max of LCS(m-1,n) and LCS(m,n-1). What about the case where both characters don't belong to the longest subsequence? These 2 cases will cover that so no need to recurse on that.

Finding LCS: Recursive Formulation

Base Case:
LCS(i, 0) = ∅ for all i
LCS(0, j) = ∅ for all j

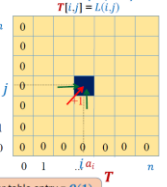
General Case:
If $a_n = b_m$ then LCS(n,m) = LCS($n-1$,$m-1$) :: a_n
If $a_n \neq b_m$ then LCS(n,m) = *bigger* of LCS(n,$m-1$) or LCS($n-1$,m)

Optimal substructure
An optimal solution to a problem (instance) contains optimal solutions to subproblems

Simplified Problem: Find the length of LCS
Let $L(n,m)$: Length of LCS of $A[1..n]$ and $B[1..m]$
$L(n,m) = 0$ if n or m is 0
If $a_n = b_m$ then $L(n,m) = L(n-1,m-1) + 1$
If $a_n \neq b_m$ then $L(n,m) = \mathbf{Max}(L(n,m-1), L(n-1,m))$

Dynamic Programming algorithm for $L(n,m)$

```
L(n,m)
{ For (i = 0 to n)  T[i,0] ← 0;
  For (j = 0 to m)  T[0,j] ← 0;
  For (j = 1 to m){
    For (i = 1 to n){
      If a_i = b_j then
          T[i,j] ← T[i – 1,j – 1] +1;
      Else {
          l1 ←T[i – 1,j];
          l2 ←T[i,j – 1];
          T[i,j] ← Max(l1, l2);
} } }
```



Time per table entry = $O(1)$
Total time = $O(nm)$

Cut & Paste Argument

1. Make a claim about the optimal substructure of the problem
2. Assume that the solution is not made up of the most optimal subproblem. Then, there is a more optimal subproblem T' that exists
3. Show that we can take T' as the subproblem to get an overall better solution, a contradiction

Improvements to LCS

→ Current space complexity is O(mn) to store the table
→ We can reduce the space requirement to O(min{m,n}), since at every iteration, we are doing either LCS(m-1,n-1)/LCS(m-1,n)/LCS(m,n-1), so, we only need the current row and previous row to perform our calculations.
→ it is min {m,n} because dp solution can work by going row then col or col then row.

Knapsack Problem

→ "What is the maximum value you can get with a subset of weights not exceeding W kg"

KNAPSACK
Input:
$(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$, and W
Output: A subset $S \subseteq \{1, 2, \ldots, n\}$ that maximizes
$\Sigma_{i \in S} v_i$ such that $\Sigma_{i \in S} w_i \leq W$

2^n subsets, so naïve algorithm is too costly!

→ 2 cases: Either we "take or don't take" the current item.
→ So, given a problem like knapsack(n, W), maxValue = max{knapsack(n-1, W), v_i + knapsack(n-1, W-w_i)
→ Have a table T of items 0 to n by weights 0 to W.
→ Initialize all T[i][0] to be 0, all T[0][j] to be 0.
→ iterate through and save the value of T[i][j] = max{T[i-1][j], V[i] + T[i-1][j – W[i]]}

$$m[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{m[i-1,j-w_i] + v_i, m[i-1,j]\} & \text{if } w_i \leq j \\ m[i-1,j] & \text{otherwise} \end{cases}$$

$v[1..5] = \{4, 2, 10, 1, 2\}$
$w[1..5] = \{12, 1, 4, 1, 2\}$
$W = 15$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	0	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	0	2	2	2	10	12	12	12	12	12	12	12	12	12	12
4	0	2	3	3	10	12										
5	0															

Pseudocode

```
(Recursive cases)                          Time per table entry = O(1)
                                           Total time = O(nW)
for i = 1, ..., n:
    for j = 0, ..., W:
        if j ≥ w[i]:
            m[i, j] ← max(m[i − 1, j − w[i]] + v[i], m[i − 1, j])
        else:
            m[i, j] ← m[i − 1, j]
```

→ Use the condition if j >= W[i] to filter out the possibility of j – w[i] going negative.
→ Total time = O(nW), space = O(nW)

Coin Change
We have n cents and need to get change in terms of denominations d1,d2..,dk. Use as little coins as possible to reach value.

Example: Changing Coins

Optimal substructure: Suppose $M[j] = t$, meaning that $j = d_{i_1} + d_{i_2} + \cdots + d_{i_t}$ for some $i_1, \ldots, i_t \in \{1, \ldots, k\}$. Then, if $j' = d_{i_1} + d_{i_2} + \cdots + d_{i_{t-1}}$, $M[j'] = t - 1$, because otherwise if $M[j'] < t - 1$, by **cut-and-paste** argument, $M[j] < t$.

$$M[j] = \begin{cases} 1 + \min_{i \in [k]} M[j - d_i], & j > 0 \\ 0, & j = 0 \\ \infty, & j < 0 \end{cases}$$

→ At each value, iterate through all coins, and save the value at M[j] as 1 + min(M[j-di]), and return M[n]

```
NUM-COINTS-DP(n, d):
    for j = 0, ..., n:
        M[j] ← ∞                          Running time = O(nk)
    M[0] ← 0
    for j = 1, ..., n:
        for i = 1, ..., k:
            if (j − d_i ≥ 0) ∧ (M[j − d_i] + 1 < M[j]):
                M[j] ← M[j − d_i] + 1
    return M[n]
```

Greedy Algorithms
- Only one subproblem at each step

Fractional Knapsack

Input:
$(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$ and W

Output:
Weights $x_1, \ldots, x_n$ that maximize $\sum_i v_i \left(\frac{x_i}{w_i}\right)$ subject to: (fraction of item i taken)
$\sum_i x_i \le W$ and $0 \le x_j \le w_j$ for all $j \in \{1, 2, \ldots, n\}$.

→ Optimize to take for the item that gives the most value per unit weight
→ Optimal Substructure: if I remove w kg of one item j from the optimal knapsack, then the remaining load must be the optimal knapsack weighing at most W – w kg that one can take from the n-1 original items and Wj – w kgs of item j.
1. **Cut and Paste**
→ Can argue using cut and paste, suppose the subproblem is not optimal, meaning there is a more optimal solution, then we can take that solution and add the Wj kg of that item j to get a more optimal solution, a contradiction.

2. **Greedy Choice Property**
Let j* be an item with the maximum value/kg, vj/wj. Then, there exists an optimal knapsack containing at least min(W, Wj*) kgs of item j*
→ argue that it "doesn't hurt" to take the greedy choice.
→ Suppose there is an optimal knapsack contains x1 kg of item 1, x2 kg of item 2… xn kg of item n such that x1 + x2 + … + xn = min(W, Wj*).
→ Replace these weights with min(W, Wj*) of item j*, weight does not change, and total value either stays the same or decreases since value/kg of j* is the maximum, so solution stays optimal

Strategy for Greedy Algorithm

- Use greedy-choice property to put $\min(w_{j^*}, W)$ kgs of item j^* in knapsack.
- If knapsack weighs W kgs, we are done.
- Otherwise, use optimal substructure to solve subproblem where all of item j^* is removed and knapsack weight limit is $W - w_{j^*}$.

Iterative greedy algorithm

```
ITER-FRAC-KNAPSACK(v, w, W):
    valperkg ← [1, 2, ..., n]
    Sort valperkg using comparison operator ≼ where i ≼ j if  v[i]/w[i] ≤ v[j]/w[j]
    for i = n to 1:
        if W = 0: break
        j ← valperkg[i]
        k ← min(w[j], W)
        print "k kgs of item j"
        W ← W − k
    return
```

→ Time: O(nlogn)
→ In greedy algorithms, might have to do some form of sorting because we want the most optimal (which is usually some max or min)

Paradigm for greedy algorithms
1. Cast problem where we have to make a choice and are left with 1 subproblem to solve
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so the greedy choice is safe
3. Use optimal substructure to show that we can combine an optimal solution to the subproblem with the greedy choice to get an optimal solution to the original problem

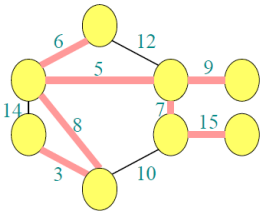
Greedy Algorithm: Minimum Spanning Trees
- MSTs are trees that connects all vertices and have the minimum weight.

Input: A connected, undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$.
- For simplicity, assume that all edge weights are distinct.

Output: A *spanning tree* T — a tree that connects all vertices — of minimum weight:

$$w(T) = \sum_{(u,v) \in T} w(u, v).$$

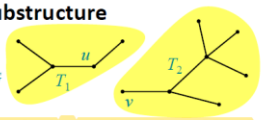
Example of MST



1. Optimal Substructure

Optimal substructure

MST T: (Other edges of G are not shown.)



Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2.
Theorem. The subtree T_1 is an MST of $G_1 = (V_1, E_1)$, the subgraph of G *induced* by the vertices of T_1:
V_1 = vertices of T_1,
$E_1 = \{(x, y) \in E : x, y \in V_1\}$.
Similarly for T_2.

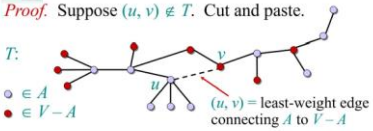
- Proof using cut and paste:
w(T) = w(u,v) + w(T1) + w(T2)
If there were a more optimal MST T1' with a lower weight, then T' = {(u,v) U T1'} U T2 would result in an even more optimal MST, a contradiction

2. Greedy choice property

Theorem. Let T be the MST of $G = (V, E)$, and let $A \subseteq V$. Suppose that $(u, v) \in E$ is the least-weight edge connecting A to $V - A$. Then, $(u, v) \in T$.

Proof of theorem

Proof. Suppose $(u, v) \notin T$. Cut and paste.



Consider the unique simple path from u to v in T.
Swap (u, v) with the first edge on this path that connects a vertex in A to a vertex in $V - A$.
A lighter-weight spanning tree than T results. □

→Build spanning tree by taking the cheapest edge connecting A to V-A

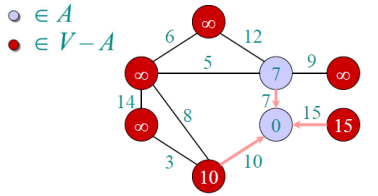
Prim's algorithm

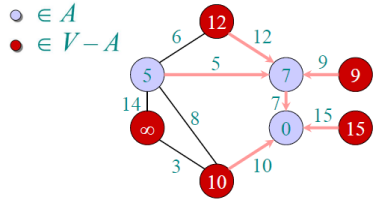
IDEA: Build the tree one vertex at a time. At each step, add a least-weight edge from the tree to some vertex outside the tree.

```
Q ← V
key[v] ← ∞ for all v ∈ V
key[s] ← 0 for some arbitrary s ∈ V
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
        for each v ∈ Adj[u]
            do if v ∈ Q and w(u, v) < key[v]
                then key[v] ← w(u, v)        ▷ DECREASE-KEY
                     π[v] ← u
```

At the end, $\{(v, \pi[v])\}$ forms the MST.

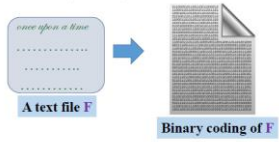
→ Q is the set of all nodes initially, start with one random node with cost 0, for each of its neighbors, if the weight of the edges is lower than key[v], decrease the value in key (i.e. decrease the value of the node), and add the cheapest node to the vertex set.





Huffman Code

Binary coding
Alphabet set A: $\{a_1, a_2, \ldots, a_n\}$
A text file: a sequence of alphabets



Question: How many bits needed to encode a text file with m characters?
Answer: $m \cdot \lceil \log_2 n \rceil$ bits.

→ n - size of alphabets, i.e. 26 for a-z. if we only have abcd, we need 2 bits, abcde needs 3 bits

Question: What is a **fixed length** coding of A?
Answer: each alphabet ← a unique binary string of length $\lceil \log_2 n \rceil$.

Question: How to decode a **fixed length binary** coding?
Answer: **Easy** ☺

0100|10100000|1011|...

→ For fixed length coding we decode by splitting entire encoding into 4 bits by 4 bits and decode letter by letter.

→ We cannot use less bits to store alphabet set, but when we store a file, perhaps we can
→ Note that there is a huge variation in the frequency of alphabets in a text, e.g. e is more common than x.

More frequent alphabets ← coding with **shorter bit string**
Less frequent alphabets ← coding with **longer bit string**

Variable Length encoding

Alphabets	Frequency f	Encoding γ
a	0.45	0
b	0.18	100
c	0.15	110
d	0.12	101
e	0.10	111

Average bit length per symbol using γ:

$$ABL(\gamma) = \sum_{x \in A} f(x) \cdot |\gamma(x)|$$
$$= 0.45 \times 1 + 0.18 \times 3 + (0.15 + 0.12 + 0.10) \times 3$$
$$= 2.1$$

→ Idea: we want to reduce the number of bits used. Take note that we also need each encoding to not be a prefix of another encoding. For example, if b was just 10 instead, then if we have 1010, we wont know if its bb or da when decoded

Prefix Coding

A coding $\gamma(A)$ is called prefix coding if there do not exist $x, y \in A$ such that

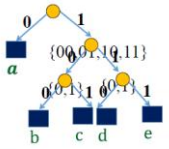
$\gamma(x)$ is prefix of $\gamma(y)$

Problem: Given a set A of n alphabets and their frequencies, compute coding y such that y is prefix coding and ABL(y) is minimum
→ Idea: use of a labelled binary tree, where leaf nodes are alphabets. The code of an alphabet is the label of path from root.

Variable length Coding

Alphabets	Frequency f	Encoding γ
a	0.45	0
b	0.18	100
c	0.15	110
d	0.12	101
e	0.10	111

Question:
How to build the labeled tree for a prefix code ?
{0, 100, 101, 110, 111}



Prefix code and Labelled Binary tree

Theorem:
For each prefix code of a set A of n alphabets, there exists a binary tree T on n leaves such that
- There is a bijective mapping between the **alphabets** and the **leaves**.
- The label of a path from root to a leaf node corresponds to the prefix code of the corresponding alphabet.

Question: Can you express **Average bit length** of γ in terms of its binary tree T ?

$$ABL(\gamma) = \sum_{x \in A} f(x) \cdot |\gamma(x)|$$
$$= \sum_{x \in A} f(x) \cdot |\text{depth}_T(x)|$$

→ sum of frequency * length of bit
→ bijective mapping: one to one function, i.e. one leaf = one alphabet

Observations on binary tree of an optimal prefix coding

→ The binary tree corresponding to optimal prefix coding must be a full binary tree where every internal node has degree exactly 2.

Question: What next?

We need to see the influence of frequencies on an optimal binary tree.

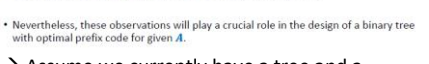
Let $a_1, a_2, \ldots, a_n$ be the alphabets of A in non-decreasing order of their frequencies.

Intuitively, more frequent alphabets should be closer to the root and less frequent alphabets should be farther from the root.

But how to organize them to achieve optimal prefix code ?

- We shall now make some simple observations about the structure of the binary tree corresponding to the optimal prefix codes.
- These observations will be about some local structure in the tree.
- Nevertheless, these observations will play a crucial role in the design of a binary tree with optimal prefix code for given A.

→ Assume we currently have a tree and a frequency map such that ai is more frequent than a1 (i.e. f(ai) > f(a1)) and its associated encoding is longer i.e.(|y(ai)| > |y(a1)|). If we swap them, can we get a lower ABL?



An important observation

Lemma: There *exists an optimal* prefix coding in which a_1 and a_2 appear as siblings in the corresponding labeled binary tree.

Important note: It is inaccurate to claim that "In every optimal prefix coding, a_1 and a_2 appear as siblings in the labeled binary string."

→ What happens after you match the 2 least frequent letters, and want to continue building the tree? we can sum up the frequencies of a1 and a2 and treat them as 'one character' with respect to its frequencies, and try to build continue applying the lemma to get the optimal encoding length for each character.

Reductions & Intractability

- Solving a problem by reducing it to another problem, solve it using that problem and getting a solution based on that

Matrix multiplication and squaring

MAT-MULTI	MAT-SQR
Input:	**Input:**
❑ Two $N \times N$ matrices A and B	❑ One $N \times N$ matrix C
Output:	**Output:**
❑ $A \times B$	❑ C^2

→ Mat-square reduces to Mat-multi, just substitute A = C and B = C into Mat-multi and we can get answer for matsquare → O(n) time reduction in copying the inputs of mat-sqr.

→ Mat multi also reduces to mat square

Claim: MAT-MULTI reduces to MAT-SQR.

Proof: Given input matrices A and B:

Construct:

$$C = \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}$$

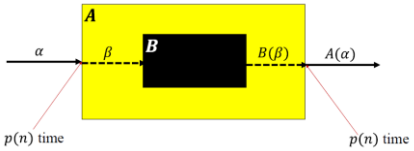
Call MAT-SQR to get

$$C^2 = \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix} \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix}$$

p(n)-time reduction

→ We say that there is a p(n)-time reduction from A to B if for any instance a of problem A of size n:

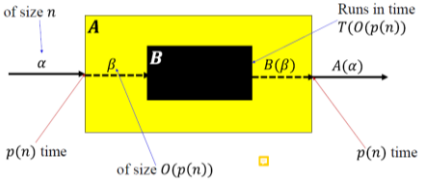
→ An instance b for problem B can be constructed in p(n) time, a solution to problem A for input a can be recovered from the solution we got from problem B to input b in p(n) time



Running Time Composition

Claim: If there is a $p(n)$-time reduction from problem A to problem B, and there exists a $T(n)$-time algorithm to solve problem B on instances of size n, then there is a $T(O(p(n)) + O(p(n))$ time algorithm to solve problem A on instances of size n.

Running Time Composition



Polynomial Time Reduction

Definition:

$$A \le_P B$$

if there is a $p(n)$-time reduction from A to B for some polynomial function $p(n) = O(n^c)$ for some constant c.

→ If B is easy, A is easy => If B has polynomial time algorithm, so does A, because we can reduce A to B and solve A in poly time

→ **A note on encoding: When we say runtime is polynomial, we refer to the length of the encoding of the problem instance**

- For polynomial time, we mean that the runtime is polynomial in the **length of the encoding of the problem instance.**
- For many problems, can use a "standard" encoding.
  - Binary encoding of integers
  - For mathematical objects (graphs, matrices, etc.): list of parameters enclosed in braces, separated by commas

→ Knapsack is not polynomial, fractional-knapsack is.

The input for both problems is a list $(v_1, w_1), \ldots, (v_n, w_n), W$. The input size is $O(n \log M + \log W)$ where M is an upper bound on the v_i's and w_i's.

→ Input size: each v,w is logM size, there are n (vi,wi) pairs. W's encoding: log W

Running time for KNAPSACK is $O(nW \log M)$. Running time for FRACTIONAL KNAPSACK is $O(n \log n \log W \log M)$.

→ Running time of knapsack with respect to its input is O(nW). With respect to encoding, n is basically nlogM, weight is a logW encoding and the runtime is W => O(nWlogM)

→ Fractional knapsack running time is O(nlogn), which is basically O(nlogn * logM * logW)

Pseudo-Polynomial Algorithms

→ Knapsack is a pseudo-polynomial algorithm

→ An algorithm that runs in polynomial time in the **numeric value of the input (e.g. the weight in knapsack)** but is exponential in the length of the input is called a pseudo-polynomial time algorithm.

Reductions

- If you have a polynomial-time reduction from A to B and you also have a polynomial-time algorithm for B, then you get a polynomial-time algorithm for A.

→ **If B is easy, A is also easy**

→ **If A is hard, B is also hard (Intuition:** A is a special case of B. If A is hard and B is easy, then A is also easy, a contradiction)

Decision Problems

→ A function that maps an instance space I to the solution set {YES, NO}

Decision vs Optimization Problems

- **Decision Problem**: Given a directed graph G with two given vertices u and v, is there a path from u to v of length $\le k$?
- **Optimization Problem**: Given a directed graph G with two given vertices u and v, what is the length of the shortest path from u to v?

Given an instance of the optimization problem and a number k, ask for a solution with value $\le k$?

Examples: a minimum spanning tree with weight $\le k$, a longest common subsequence with length $> k$, a knapsack solution with value $> k$, etc.

→ **Decision problems reduces to optimization problems!**

→ Decision problem is no harder than the optimization problem. (If B is easy A is easy so A cant be "harder" than B?) Given the value of the optimal solution, simply check whether it is <= k.

→ If we cant solve the decision problem quickly, we cant solve the optimization problem quickly. So, we can just study decision problems for hardness

Reductions between Decision Problems

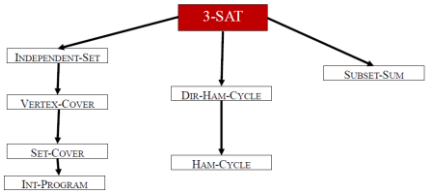
Given two decision problems A and B, a **polynomial-time reduction** from A to B, denoted $A \le_P B$, is a transformation from instances α of A to instances β of B such that:

1. α is a YES-instance for A **if and only if** β is a YES-instance for B.
2. The transformation takes polynomial time in the size of α.

→ **Show that the transformation takes poly-time**

→ **Show that yes instance of A is yes instance of B, show that yes instance of B is yes instance of A**

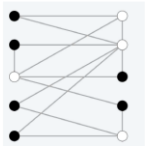
Reductions



Independent Set <=p Vertex Cover

Independent Set:

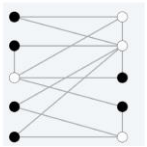
Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or more) vertices such that no two are adjacent?



Independent set of size 6

Vertex Cover:

Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or fewer) vertices such that each edge is incident to at least one vertex in the subset?



Vertex cover of size 4

→ Since we are trying to show that independent set <=p vertex cover, we check whether G has a vertex cover of size n – k

→ Reduction clearly runs in poly time.

Reduction: To check whether G has an independent set of size k, we check whether G has a vertex cover of size $n - k$.

Proof:

- Suppose (G, k) is a YES-instance of INDEPENDENT-SET. So, there is a subset S of size $\ge k$ that is an independent set.
- **Claim**: $V - S$ is a vertex cover, of size $\le n - k$. Why?
- Let $(u, v) \in E$. Then, either $u \notin S$ or $v \notin S$. So, either u or v in $V - S$. Done!

→ We take any edge with ends (u,v), since S is an independent set, either u or v is not in S and is in V-S. So V-S is a vertex cover

Reduction: To check whether G has an independent set of size k, we check whether G has a vertex cover of size $n - k$.

Proof:

- Suppose $(G, n - k)$ is a YES-instance of VERTEX-COVER. So, there is a subset S of size $\le n - k$ that is a vertex cover.
- **Claim**: $V - S$ is an independent set, of size $\ge k$. Why?
- Let $(u, v) \in E$ with both u and v in $V - S$. But then, S does not cover (u, v), a contradiction!

→ We have a set S of size <= n-k that is a vertex cover. We claim that we have a set V-S of size >= k that is a independent set.

→ Proof is that we take any edge (u,v) in E, If both u and v are in independent set, then S is no longer a vertex cover, a contradiction.

Vertex Cover <=p Set Cover

Given integers k and n, and a collection S of subsets of $\{1, \ldots, n\}$, are there $\le k$ of these subsets whose union equals $\{1, \ldots, n\}$?

$S_1 = \{3,7\}$ $S_4 = \{2,4\}$
$S_2 = \{3,4,5,6\}$ $S_5 = \{5\}$
$S_3 = \{1\}$ $S_6 = \{1,2,6,7\}$
$k = 2, n = 7$

→ Answer is S2 and S6

Reduction:

- Given (G, k) instance of VERTEX-COVER, we generate an instance $(n, k', \mathcal{S})$ of Set-Cover.
- Set $n = |E(G)|$, and $k' = k$.
- Order the edges of G arbitrarily: $e_1, \ldots, e_n$. For each $v \in V(G)$: $S_v = \{i : e_i \text{ incident on } v\}$
  $\mathcal{S}$ is the collection of all such subsets S_v.

1. Transformation

→ k = k'. We construct the graph by taking the set cover input (i.e. the sets, value k, value n), transform into vertex cover by:

1. Set n = number of edges in vertex cover
2. Set k' = k in vertex cover
3. Set each set as a node in vertex cover

→ Reduction takes poly time

2. Yes(Vertex Cover) => Yes(Set Cover)

Proof: Yes(Vertex Cover) => We have a set S of size <= k that covers all edges in the graph. Since each edge represents a number in set cover, we have a set cover <= k by definition

Suppose (G, k) is a YES-instance of VERTEX-COVER. Let U be the subset of size $\le k$ forming the vertex cover. Then, by definition, the union of S_u's over all $u \in U$ is $\{1, \ldots, n\}$.

3. Yes(Set Cover) => Yes(Vertex Cover)
Proof: Yes(Set Cover) => we have a subset S of size <= k such that the intersection of the sets have all the values from 1 to n. Since we represent the values to be edges and sets to be nodes, then we have <= k nodes where all the edges are covered, forming a vertex cover of size <= k by definition.

Suppose $(n, k, \mathcal{S})$ is a YES-instance of SET-COVER. Let the cover correspond to the sets $S_{v_1}, \ldots, S_{v_t}$ for $t \leq k$. Then, the vertices $v_1, \ldots, v_t$ form a vertex cover in G.

Satisfiability (SAT problem) - Terminology

- **Literal**: A Boolean variable or its negation. $x_i, \overline{x_i}$
- **Clause**: A disjunction (OR) of literals. $C_j = x_1 \vee \overline{x_2} \vee x_3$
- **Conjunctive Normal Form (CNF)**: a formula Φ that is a conjunction (AND) of clauses $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$
- **SAT**: Given a CNF formula Φ, does it have a satisfying truth assignment?

3-SAT:
→ SAT where each clause contains exactly 3 literals (that are not necessarily distinct)
→ For SAT, as long as there is a satisfying assignment we return YES for that CNF

$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$
Satisfying assignment: $x_1 = \text{True}, x_2 = \text{True}, x_3 = \text{False}, x_4 = \text{True}$
Non-satisfying assignment: $x_1 = \text{True}, x_2 = \text{False}, x_3 = \text{False}, x_4 = \text{False}$

Φ is a YES-instance if and only if it admits at least one satisfying assignment.

3SAT <=p Independent Set
1. Reduction

Reduction
- G contains 3 vertices for each clause, one for each literal
- Connect 3 literals in clause in a triangle
- Connect literal to each of its negations
- Set k = number of clauses

$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$

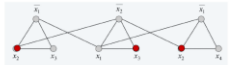
→ Reduction runs in poly time

2. Yes(3-sat) => Yes(independent-set)
Yes instance of 3-sat → there is a satisfying assignment of the CNF → representing each literal as a node, where each vertex that is true is in the independent set, we have a set of vertices of size k that do not share an edge with each other → yes instance of independent set

Suppose Φ is a YES-instance. Take any satisfying assignment for Φ and select a true literal from each clause. Corresponding k vertices form an independent set in G.

3. Yes(independent-set) => Yes(3-sat)
Yes instance of independent-set → There is a set S of edges (u,v) where either u or v is in S → set each vertex in S to be a true literal → since one clause will have at most one vertex in S and each clause will at least have one vertex in S → yes instance of independent set

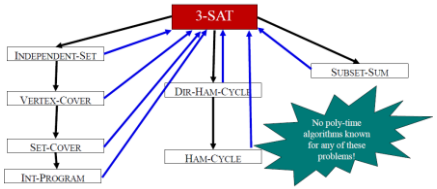
Reduction
- G contains 3 vertices for each clause, one for each literal
- Connect 3 literals in clause in a triangle
- Connect literal to each of its negations
- Set k = number of clauses



$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$

Suppose (G, k) is a YES-instance. Let S be the independent set of size k. Each of the k triangles must contain exactly one vertex in S. Set these literals to true, so all clauses satisfied.

NP-Completeness



→ Based on this, if 3-SAT is hard, the rest of the problems are also hard (i.e. if 3-sat does not have poly-time algo, none of these problems have poly-time algorithms)
→ Moreover, since all these problems are np-complete, if any of these problems are hard, all of the problems are hard

NP
- NP (or non-deterministic polynomial) is the class of problems for which polynomial-time verifiable certificates of yes-instances exist

E.g. **Subset Sum**

Recall: In SUBSET-SUM, given a list of integers S and target t, problem is to decide if there is $S' \subseteq S$ that sums up to t.

Certificate is the subset S' itself. Verifier checks whether the sum of elements of S' is t in polynomial time.

→ Subset Sum is in NP
E.g. Ham Cycle

Recall: In Ham-Cycle, given a graph G, problem is to decide whether there is a simple cycle that visits each vertex once.

Certificate is the cycle itself. Verifier checks in polynomial time whether it is a cycle and visits each vertex once.
→ Hamiltonian Cycle is in NP

P is a subset of NP
→ Why? To verify the answer, just run the algorithm in P time. We have a verifiable certificate of a yes instance of the problem in poly time

NP-Hard and NP-complete
NP-Hard: For every problem B in NP: B <=p A (All np problems can reduce to the current problem)
NP-Complete: Problem A is said to be NP-complete if it is in NP and NP-hard

Cook-Levin Theorem
Theorem: Any problem in NP reduces to 3-sat in poly time. Therefore, 3-sat is NP and NP-hard i.e. NP complete

Theorem: *Any* problem in **NP** poly-time reduces to 3-SAT. Hence, 3-SAT is **NP**-hard and **NP**-complete.

Hence, so are HAM-CYCLE, SUBSET-SUM, VERTEX-COVER, INDEPENDENT-SET,…

P=NP
→ We don't know if P=NP or P=/=NP.

Showing NP-Completeness
- Show that X is in NP
- Show that X is in NP-hard, by giving a poly-time reduction from **A to X (not the other way round)**

- To show that the reduction is valid, you need to show that
  - The reduction runs in polynomial time (if this is clear, you can simply say that it is clear)
  - If the instance of A is a YES-instance, then the instance of X is also a YES-instance
  - If the instance of X is a YES-instance, then the instance of A is also a YES-instance

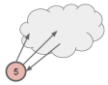
Graph tips

Degree of a Graph

Indegree = Incident edges that are **incoming**
Outdegree = Incident edges that are **outgoing**

e.g.

Indegree of this vertex = 1
Outdegree of this vertex = 2



Handshaking Lemma

The **total degree of graph G** is the sum of the degrees of all vertices of G.

In an undirected graph G, the total degree of G is **twice** the number of **edges**:

$$\sum_{v \in V} deg(v) = 2|E|$$

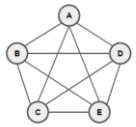
Note: Keep this lemma in mind! I have had to use this lemma to prove some graph properties or to analyse runtime of algorithms!

Complete Graph

Clique / Complete Graph: All pairs connected by edges

How many edges are there if there are n nodes?

$$\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$$



Combinatorics

→ Combination, nCm
(nCk) = n! / k!(n-k)!
→ Permutation, nPm (order matters)
nPk = nCk * kPk = k * nCk
→ Inclusion Exclusion Principle

[2 sets] Let $A, B \subset S$. Then $P(A \cup B) = P(A) + P(B) - P(A \cap B)$.

[3 sets] Let $A, B, C \subset S$. Then $P(A \cup B \cup C) = P(A) + P(B) + P(C) - P(A \cap B) - P(A \cap C) - P(B \cap C) + P(A \cap B \cap C)$.

[4 sets] Let $A, B, C, D \subset S$. Then $P(A \cup B \cup C \cup D) = P(A) + P(B) + P(C) + P(D) - P(A \cap B) - P(A \cap C) - P(A \cap D) - P(B \cap C) - P(B \cap D) - P(C \cap D) + P(A \cap B \cap C) + P(A \cap B \cap D) + P(A \cap C \cap D) + P(B \cap C \cap D) - P(A \cap B \cap C \cap D)$.

Bayes Theorem

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

The following may sometimes be useful when we do not have access to $P(B)$ directly.

$$P(A_i \mid B) = \frac{P(B \mid A_i)P(A_i)}{P(B \mid A_1)P(A_1) + \cdots + P(B \mid A_n)P(A_n)}$$

Independence

Let A and B be 2 events. We say they are independent when the probability of one event does not affect the probability of the other, e.g. coin toss
=> P(A intsct B) = P(A)P(B | A) = P(A)P(B)

$\binom{n}{k} \leq (\frac{en}{k})^k$ and $k! \geq (\frac{k}{c})^k$.

Time Complexity/Master Theorem

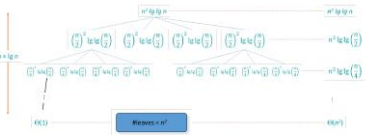
T(n) = 2T(n-1) + Θ(1) = Θ(2^n)

2. $T(n) = T(n-1) + T(n-2) + \Theta(1)$
Ans: Note that, $T(n) = F(n) + c = \Theta(\Phi^n) = \Theta((\frac{\sqrt{5}+1}{2})^n)$

→ Fibonacci number, just memorize
lglgn < lgn < n < n^2 < n^3 < 2^n
ln(n) = Θ(log(anybase(n))
→ Any (lgn)^x is definitely o(n^y)
→ Any n^x is confirm o(2^n)
→ lg(n!) = O(nlgn)
→ n^2lg((lgn)!) = Θ(n^2lgnlglgn)

Soln: $2^{2 \cdot 2^{\lg \lg n}} \ll n^2 \lg \lg n \ll n^3 \equiv \sum_{i=2}^{n} \frac{n^3}{i(i-1)} \ll n^{\lg n} \ll 2^n \ll (\lg n)^n \ll n!$

For (v), T(n) = 4 T(n/2) + n^2 lg lg n



$T(n) = \sum_{i=0}^{\lg n-1} n^2 \lg \lg \left(\frac{n}{2^i}\right) = n^2 \sum_{i=0}^{\lg n-1} \lg(\lg n - i) = n^2 \lg((\lg n)!) = \Theta(n^2 \lg n \lg \lg n)$

Differentiate lglgn:

$$y' = \frac{\frac{1}{x}}{\log x} = \frac{1}{x \cdot \log x}$$

Differentiate ln(x): d/dx(ln(x)) = 1/x

→ √n recursion gives lglg(n) time complexity
T(n) = 2T(√n) + 1 = Θ(lg(n)) by recursion tree
T(n,m) = T(n/2,m) + m = Θ(mlgn) by recursion tree
T(n) = (√n + 1)T(√n) + √n = O(n) by substitution method

6. $T(n) = (\sqrt{n}+1)T(\sqrt{n}) + \sqrt{n}$
Ans: Use substitution for this question, the following shows us how to get the upper bound O, nc that we can similarly show the lowerbound.

- Step 1: Guess $T(n) = O(n)$.
- Step 2: We aim to show that $T(n) \leq cn - c - 1$
Base case: Let $n_0 = 2$, $T(2) = q$, set c such that $q \leq 2c - c - 1 = c - 1$
Inductive case: By strong induction, we assume that $T(i) \leq ci - c - 1$ $T(n)$ where $2 \leq i <$
Let us consider $i = n$ case:

$$\begin{aligned} T(n) &= (\sqrt{n}+1)T(\sqrt{n}) + \sqrt{n} \\ &\leq (\sqrt{n}+1)(c\sqrt{n} - c - 1) + \sqrt{n} \\ &\leq cn - c\sqrt{n} - \sqrt{n} + c\sqrt{n} - c - 1 + \sqrt{n} \\ &\leq cn - c - 1 \end{aligned}$$

Telescoping Method:

(iv) T(n) = 4 T(n/4) + n lg lg lg n

We can solve it using telescoping. Dividing both sides of equation (i) by n, we have:

$\frac{T(n)}{n} = \frac{T(n/4)}{n/4} + \lg\lg\lg n$.
By telescoping, we also have:
$\frac{T(n/4)}{n/4} = \frac{T(n/16)}{n/16} + \lg\lg\lg \frac{n}{4}$.
$\frac{T(n/16)}{n/16} = \frac{T(n/64)}{n/64} + \lg\lg\lg \frac{n}{16}$.
$\frac{T(n/64)}{n/64} = \frac{T(n/256)}{n/256} + \lg\lg\lg \frac{n}{64}$.
...
$\frac{T(4)}{4} = \frac{T(1)}{1} + \lg\lg\lg 4$.

By summing over all equations, we have:
$\frac{T(n)}{n} = \frac{T(1)}{1} + \lg\lg\lg n + \lg\lg(\lg n - 2) + \lg\lg(\lg n - 4) + \lg\lg(\lg n - 6) \ldots + \lg\lg\lg 4$.
$\frac{T(n)}{n} = \frac{T(1)}{1} + \Theta(\lg n \lg\lg\lg n)$.

Hence, $T(n) = \Theta(n \lg n \lg\lg\lg n)$.

Binary Multiplication

Suppose we wish to multiply two four-bit numbers, 1011 and 1010:

	1 0 1 1	this is 11_{10}
×	1 0 1 0	this is 10_{10}
	0 0 0 0	1011 x 0
	1 0 1 1	1011 x 1, shifted one position to the left
	0 0 0 0	1011 x 0, shifted two positions to the left
+	1 0 1 1	1011 x 1, shifted three positions to the left
	1 1 0 1 1 1 0	this is 110_{10}

Question 3: Consider the function Fun(x, y). What is its output? What is the invariant for the wl loop? Can you show that this algorithm correctly compute the output?

```
Fun(x, y):
1. ans = 0, p = x, q = y
2. While q > 0 do
   (a) r ← q mod 2
   (b) q ← q/2
   (c) ans ← ans + r × p
   (d) p ← p × 2
3. return ans
```

Solution: The output of the function is $x \times y$.
Let $b_0 b_{k-1} \ldots b_2 b_1 b_0$ be the binary representation of y. Then, the binary representation of $y/2^i$ is $b_k \ldots b_i$ and the binary representation of $y \mod 2^i$ is $b_{i-1} \ldots b_1 b_0$.
The invariant is: For the i-th iteration, q is $y/2^i$, p is $x \times 2^i$, and ans is $x \times (y \mod 2^i)$.
Initiation: In the 0-th iteration, $i = 0$. We have q is y, p is x and ans is $x \times (y \mod 2^0) = 0$. This is exactly the assignment in step 1.
Maintenance: After the i-th iteration, q is $y/2^i$, p is $x \times 2^i$, and ans is $x \times (y \mod 2^i)$. Then, the $(i+1)$-th iteration goes through steps 3-5. After step 3, r is $q \mod 2 = (y/2^i) \mod 2 = b_i$. q is set to be $q/2 = (y/2^i)/2$, which is $y/2^{i+1}$.
After step 4, $ans = x \times (y \mod 2^i) + b_i \times x \times 2^i = x \times (y \mod 2^i + b_i 2^i) = x \times (y \mod 2^{i+1})$
After step 5, p is set to be $p \times 2 = x \times 2^{i+1}$.
In summary, after the $(i+1)$-th iteration, q is $y/2^{i+1}$, p is $x \times 2^{i+1}$, and ans is $x \times (y \mod 2^{i+1})$.
Termination: The while loop terminates when $q = 0$, which is the $(k+1)$-th iteration. According to the invariant, ans is $x \times (y \mod 2^{k+1}) = x \times y$. The algorithm computes the correct output.

Question 4: Given an array A of integers, a pair (i, j) is said to be an *inversion* if $i < j$ and $A[i] > A[j]$. Design an $O(n \log n)$ time algorithm that counts the number of inversions in a given array of size n.

→ Sort the array, have a pointer to_swap initialized at the end of the sorted array. O(nlgn)
→ Iterate through the original array, binary searching for the index of A[i]. inversions += idx – 1.
→ swap current idx with to_swap. Update the new value at to_swap to be infinity. Decrement to_swap.

Mergesort Algo

```
MERGE(A[], B[])
1  i = 1, j = 1
2  A.append(∞), B.append(∞)
3  C[]
4  while i < A.length or j < B.length
5      if A[i] < B[j]
6          C[i + j − 1] = A[i], i = i + 1
7      else C[i + j − 1] = B[j], j = j + 1
8  return C[]

MERGE-SORT(A[], l, r)
1  if l = r return A[]
2  m = ⌊(l + r)/2⌋
3  leftArray = MERGE-SORT(A[], l, m)
4  rightArray = MERGE-SORT(A[], m + 1, r)
5  return MERGE(leftArray, rightArray)
```

Proving Mergesort

First assumption: the merge routine you use merges two sorted arrays into a sorted array. Second assumption: the merge routine terminates

- **Base case:** n = 1, array of 1 element is always sorted
- **Inductive hypothesis:** merge sort works for n = 1,2,...,k
- **Inductive step:** n = k+1

Now we need to prove the inductive step is correct.

Merge sort splits the array into two subarrays `L = [1,n/2]` and `R = [n/2 + 1, n]`. See that `ceil(n/2)` is smaller than `k` based on the facts above. By our inductive hypothesis both results of merge sort for L and R are correctly sorted (as they are within `[1,k]` range). Furthermore from our assumption merge routine merges them into a sorted array which contains all elements because `size(L) + size(R) = n` so this means that it correctly sorted an array of size `n = k+1`.

Question 3: Consider an undirected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. Order the vertice in V randomly. Let I be the set of vertices v whose all the neighbors occur after v in the above random order. Note, I is an independent set, i.e., no edges between the vertices in I (think why?). Show tha expected size of I is $\sum_{v \in V} \frac{1}{d(v)+1}$, where $d(v)$ denotes the degree of v in G.

Solution: Consider any arbitrary edge $(u, v) \in E$. Suppose $u \in I$. By definition of I, v occurs after i in the order. This means that $v \notin I$. Hence I is an independent set.
Let $X_v = 1\{v \in I\}$. (We use this notation $1\{v \in I\}$ to denote that $X_v = 1$ if $v \in I$; 0 otherwise, i.e. X_v is an indicator random variable denoting whether $v \in I$ or not.) Then $\mathbb{E}[X_v]$ denotes the probabilit that v occurs before all its neighbors in the order. Since v and its neighbors are ordered randomly, w have $\mathbb{E}[X_v] = \frac{1}{d(v)+1}$. By linearity of expectation, we have $\mathbb{E}[|I|] = \sum_{v \in V} \mathbb{E}[X_v] = \sum_{v \in V} \frac{1}{d(v)+1}$.

Examples:

Some hierarchies: > is "o", = is "Θ"
$2^{2^{n+1}} > 2^{2^n} > (n+1)! > n! > e^n > n \cdot 2^n > 2^n > (3/2)^n > (\lg n)^{\lg n} = n^{\lg\lg n} > (\lg n)! > n^3 > n^2 = 4^{\lg n}$
$> n \lg n = \lg(n!) > n = 2^{\lg n} > (\sqrt{2})^{\lg n} > 2^{\sqrt{2}\lg n} > \lg^2 n > \ln n > \sqrt{\lg n} > \ln\ln n > 2^{\lg^* n} > \lg^* \lg n = \lg^* n > \lg\lg^* n > 1 = n^{1/\lg n}$

Some recurrences:
Binary search: $T(n) = T(n/2) + 1 = O(\lg n)$
Linear search: $T(n) = T(n-1) + 1 = O(n)$
$T(n) = 4T(n/3) + n \lg n \Rightarrow \Theta(n^{\lg_3 4})$ (MT case 1)
$T(n) = 3T(n/3) + n/\lg n \Rightarrow \Theta(n \lg\lg n)$ (tree)
$T(n) = 4t(n/2) + n^2\sqrt{n} \Rightarrow \Theta(n^{2.5})$ (MT case 3)
$T(n) = 3T(n/3 - 2) + n/2 \Rightarrow \Theta(n \lg n)$ (bounds)
$T(n) = 2T(n/2) + n/\lg n \Rightarrow \Theta(n \lg\lg n)$ (tree)
$T(n) = T(n/2) + T(n/4) + T(n/8) + n \Rightarrow \Theta(n)$ (bounds)
$T(n) = T(n-1) + 1/n \Rightarrow \Theta(\lg n)$ (substitution)
$T(n) = T(n-1) + \lg n \Rightarrow \Theta(n \lg n)$ (bounds)
$T(n) = T(n-2) + 1/\lg n \Rightarrow \Theta(n/\lg n)$ (bounds)
$T(n) = \sqrt{n}T(\sqrt{n}) + n \Rightarrow T(n) = \Theta(n \lg\lg n)$ (tree)

Math stuff:
$\sum_{i=1}^{n} i = n(n+1)/2 = \Theta(n^2)$
$a^{\lg_b c} = c^{\lg_b a}$
$\sum_{k=1}^{n} \frac{1}{k} = \Theta(\lg n)$ – harmonic series

3. $T(n) = T(\sqrt{n}) + \Theta(1)$
Ans: Using tree recursion, $T(n) = height * c = lg(lg(n)) * c = \Theta(lg(lg(n)))$

T(n) = 2T(√n) + 1 = Θ(lg(n)) by recursion tree
T(n,m) = T(n/2,m) + m = Θ(mlgn) by recursion tree
T(n) = (√n + 1)T(√n) + √n = O(n) by substitution method

$$\sum_{x=1}^{\log_3 n} \frac{1}{x^2} = \Theta(1).$$

$$\sum_{k=1}^{n} \frac{1}{k(k+1)} = 1 - \frac{1}{n+1}$$

Some hierarchies:

$2^{2^{n+1}} > 2^{2^n} > 2^{n\log n} > n^n > (n+1)! > n! > e^n > 3^n > n.2^n > 2^n > (3/2)^n > (\lg n)^{\lg n} = n^{\lg\lg n} > (\lg n)! = 2^{\lg n \lg\lg n} > n^3 > n^2 = 4^{\lg n} > n\lg n = \lg(n!) > n\lg\lg n = \Sigma^{n-2}\lg\lg(n-i) > n = 2^{\lg n} > (\sqrt{2})^{\lg n} > 2^{\sqrt{(2\lg n)}} > \lg^2(n) > \lg n\lg\lg n = \lg((\lg n)!) > \ln(n) > \sqrt{(\lg n)} > 10^{\lg((\lg\lg n)!)/\lg\lg n} > \lg\lg n = 2^{\lg\lg\lg n} > 2^{\lg^* n} > \lg^* n > \lg\lg^* n > 1 = n^{1/\lg n}$

→ When you try logb(a) and the f(n) has a log term, check case 2 of MT, it might apply.
→ if you logb(a) and you see that the f(n) term has the same n power but with a log term (e.g. n^2lglgn) consider telescoping

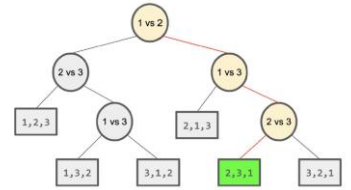
Stirling Approximation

$$\log_2(n!) = n \log_2 n - n \log_2 e + \Theta(\log_2 n).$$

Specifying the constant in the $O(\ln n)$ error term gives $\frac{1}{2}\ln(2\pi n)$, yielding the more precise formula:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n,$$

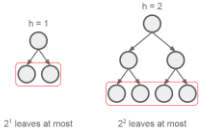
Sorting Decision Trees



Theorem: Any decision tree that can sort n elements must have height $\Omega(nlgn)$

Claim 1: A height h binary tree has ≤ 2^h leaves

Visual Idea instead of Proof:



→ See how many leaves are there in this comparison based tree (e.g. for sorting, there are n! possibilities so n! leaves, log(n!) = omega(nlgn))
→ Can be used to prove lower bound (omega), which means I need at least this amount of comparison

1. (1 point) Consider a variant of the knapsack problem with n items where there is an extra parameter $R \in \{1, 2, \ldots, n\}$, and you are supposed to choose at most R items with total weight at most W and as high total value as possible. You want to output the maximum value.

   Explain how you can modify the dynamic programming algorithm for knapsack to solve this problem (in particular, write down the recurrence relating the solution of a problem to the solutions of its subproblems), and analyze the running time of your algorithm.

1. We can add a third parameter k that indicates that the chosen set of items is of size at most k:

$$m[i,j,k] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \text{ or } k = 0 \\ \max\{m[i-1, j-w_i, k-1] + v_i,\ m[i-1, j, k]\} & \text{if } w_i \le j \\ m[i-1, j, k] & \text{otherwise} \end{cases}$$

   The second case says that if item i is chosen, then we maximize the value from choosing at most $k-1$ out of the first $i-1$ items subject to the weight constraint (which is reduced by w_i); otherwise, we simply maximize the value from choosing at most k out of the first $i-1$ items subject to the weight constraint (which is the same as before).

   The running time is $O(nWR)$. To achieve this running time, we fill in the table in the order $m[0,*,*], m[1,*,*], m[2,*,*], \ldots, m[n,*,*]$. Since the value of $m[i,j,k]$ depends only on $m[i-1,*,*]$, we have all the necessary values to compute $m[i,j,k]$. There are $O(nWR)$ entries, and computing each entry takes time $O(1)$. The final answer is then $m[n,W,R]$.

→ if don't take, we don't minus the weight and the max item indices

3. (1 point) You have n tasks to complete and n helpers. The probability that helper i can complete task j is $p_{ij} \in [0, 1]$, independently of other helpers and tasks. Each helper can only be assigned to one task (so each task must be assigned to exactly one helper).

   (a) Design and analyze an algorithm running in time $O(n \cdot 2^n)$ that computes the maximum probability of all tasks being completed, where the maximum is taken across all possible assignments.

   (b) What is the running time of a brute-force algorithm that checks all possible assignments and finds one that maximizes the probability of all tasks being completed?

1

3. Note that if you assign helper i to task h_i for each $i = 1, 2, \ldots, n$, then by independence, the probability that all tasks are completed is $p_{1,h_1} p_{2,h_2} \cdots p_{n,h_n}$. So we want to find the maximum value that this product can attain.

   (a) For a set $S \subseteq \{1, 2, \ldots, n\}$, if S has size k, let $m[S]$ be the maximum probability[1] that we can complete all tasks in S by assigning them to helpers $1, 2, \ldots, k$. We have the following recurrence:

$$m[S] = \begin{cases} 1 & \text{if } S = \varnothing \\ \max_{j \in S} \{m[S \setminus \{j\}] \cdot p_{k,j}\} & \text{otherwise} \end{cases}$$

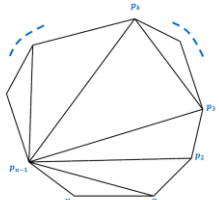
   Here, if S is empty, the probability of completing all tasks in S is trivially 1. Otherwise, we try assigning to helper k each task $j \in S$, with success probability $p_{k,j}$. The success probability of the k tasks in S is then maximized when the assignment of the tasks in $S \setminus \{j\}$ to helpers $1, 2, \ldots, k-1$ is optimal; this optimal probability is $m[S \setminus \{j\}]$.

   The running time is $O(n \cdot 2^n)$. To achieve this, we fill in the table from smaller sets S to larger ones. There are $O(2^n)$ entries, and computing each entry takes time $O(n)$. The final answer is then $m[\{1, 2, \ldots, n\}]$.

   (b) If you check all possible assignments, this would take time $\Theta(n \cdot n!)$, which is higher than the time that the dynamic programming algorithm in part (a) takes.

Triangulation of A Convex Polygon

NUS National University of Singapore



$\omega(i,j,k)$: Weight of triangle formed by p_i, p_j, p_k.

Assumption: It takes $O(1)$ time to compute $\omega(i,j,k)$

Cost of a triangulation : Sum of the weight of $n-2$ triangles formed.

```
def iter_opt_triangulation(l, n):
01.  for (i =1 to n-1):
02.      T[i, i+1] = 0 # base cases
03.  # size=k => polygon with k+1 pts
04.  for (size = 2 to n-1):
05.      # n-1-size prevents overflow of i+size
06.      for (i = 1 to n-size):
07.          j = i + size
08.          T[i, j] = INF
09.          # try triangles
10.          for (i < k < j):
11.              curr = T[i, k] + T[k, j] + w(i, k, j)
12.              # take the better result
13.              T[i, j] = min(T[i, j], curr)
14.  return T[1, n]
```

1. Start from a small polygon
2. Increase size of polygon o
   a. Note that you will need a polygons smaller than it computed
   b. Try all the polygons of s
      i. Guess all the triang that polygon

→ Start from smallest polygons. Size is the size of the polygon. So initially we iterate from i = 1, j will be 3, then i = 2, j = 4…

K-SAT-EXACT <=p (K-1)-SAT-EXACT

Part c)

Generalise your reduction in part b) such that you can reduce from k-SAT-EXACT to (k-1)-SAT-EXACT, where k ≥ 4

Answer

The idea from part b) doesn't change. But instead of taking the first two and last two, what we do is take the first (k-2) and the last (k-2) of the clause. (In part (b), we take the first two and the last two), and like before we OR it with a new variable for one clause, and OR it with the negation of that variable for the other one.

Now note that this might imply some variables are used twice. e.g. if we reduce from 5-SAT-EXACT to 4-SAT-EXACT:
(a OR b OR c OR d OR e) → (a OR b OR c OR z) AND (c OR d OR e or z), the variable c "overlaps" and will be taken twice in the pair.

Proof of correctness should still remain the same.

Online List Problem

Question 2: Let us consider the following *Online List Problem*. In this problem we are given a doubly linked list L containing n items (you may consider them to be distinct), and a sequence of m search operations $s_1, s_2, \cdots, s_m$ with the following restrictions:

- Each search operation s_i will look like: Given an item e_i you have to correctly say whether e_i is in the list L or not.
- The sequence of search operations will arrive in online fashion, i.e., at time t you will get a request for the operation s_t and you have to answer correctly. (One important thing to note, you cannot see future search queries.)
- You can access the linked list only through the HEAD pointer. So if at time[1] t you are asked to search an item e_t which is the third element in the list at that moment, then you have to pay 3 unit cost to find that element.
- During answering a search query you may update the list. However, at any point you can only swap two neighboring elements in the list, and each such swap will cost you one unit. (So during any search operation if you will perform k swaps, you will have to pay additional k unit cost.)

[1]Do not get confused between time and cost. Here I use the term *time* to identify which operation you are going to serve at that particular moment, whereas the cost captures the complexity of each operation.

Now keeping all these restrictions in mind, ideally we would like to design an algorithm that achieves minimum total cost.

Ooh, that must be a very difficult task! So to make your life simpler, I am not asking you to design an algorithm with the minimum cost. Instead I ask you to consider the following simple heuristic algorithm: Whenever we search an item in the list, if we find it we bring that in front of the list (by performing a sequence of swaps).

You may wonder since this is an extremely simple heuristic strategy, its total cost will no way be close to the minimum. However, I have a different opinion. Suppose if all the m search operations were given in advance (instead of arriving in online fashion) then the minimum cost that one (optimum algorithm) could achieve is C_{opt}. I claim that the total cost of the above heuristic algorithm is just at most $4C_{opt}$. If you answer the following series of questions, you will also be able to prove this claim.

(a) During the search operation s_t suppose you are asked to search for the item e_t. Let r and r^* be the rank of e_t in the list maintained by the heuristic algorithm (denoted by L) and the optimum algorithm (denoted by L^*) respectively. Further suppose the optimum algorithm performs t^* swaps during the search operation s_t. What is the difference between the cost to perform s_t by the heuristic and the optimum algorithm?

Answer: The cost to search the item by the optimal algorithm is r^*, and so the overall actual cost is $r^* + t^*$. Similarly the overall actual cost for the heuristic algorithm is $r + (r-1) = 2r - 1$, where r is for the search and $r-1$ for those many swaps. So the difference is $2r - r^* - t^* - 1$.

→ Difference between cost to perform search operation s_t by heuristic and optimum algorithm?
- In heuristic algorithm, the current rank of the item is r. We bring in all the way to the front of the list, so the cost of swap is r-1, total search cost = 2r-1.
- In optimum algorithm, the current rank of the item is r*. The cost of swaps is t*. So, the total search cost = r* + t*
- Difference in cost = 2r – r* - t* - 1.

(b) Let us define the *distance* between two lists L and L' containing exactly the same items (but in different order) as number of pairs (x, y) such that relative ordering of x and y in L and L' are different. For example, if $L = 10, 20, 30, 40$ and $L' = 10, 30, 40, 20$, then the distance between them is 2 (since (20, 30), (20, 40) are the pairs with different relative ordering in L and L'). Show that $\phi(t)$, defined as the twice the distance between L and L^* at time t, is a valid potential function.

Answer: At the beginning both the lists (one maintained by the optimal algorithm and another maintained by the heuristic one) are same, and so $\phi = 0$. By the definition of the distance it can never be negative and hence at any point of time $\phi \ge 0$. So ϕ is a valid potential function.

→ As long as a potential function never goes below 0 it is a valid potential function. When both lists are the same, the potential is 0. After that the distance is always positive

(c) Show that during performing s_t by the heuristic algorithm, each swap changes the distance between L and L^* by at most one.

Answer: Suppose we are swapping items e_i and e_j where e_i appears before e_j. We know that such a swap can happen only between two consecutive elements in the list. So relative ordering of only these two elements will change. Either the distance will increase by 1 (if e_i appears before e_j in the optimum list) or decrease by 1 (if e_i appears after e_j in the optimum list).

→ We can only swap adjacent elements, so swaps increases the distance between L and L* by at most 1, since by swapping adjacent elements, the relative ordering of at most 2 elements change

(d) Next prove that the change in potential from t-th time to $(t+1)$-th time is at most $4r^* + 2t^* - 2r - 2$.

Answer: Note, during each operation s_t we will move exactly one element e_t in the heuristic algorithm to the front. Since rank of e_t in the list of optimum algorithm is r^*, its movement can increase the distance by at most $r^* - 1$. Since r is the rank of e_t in the list maintained by the heuristic algorithm, this movement also leads to a decrease in distance due to change in relative ordering with at least $r - r^*$ many elements (which appears before e_t in the heuristic algorithm list but after the optimum algorithm list). t^* swaps by the optimum algorithm can increase the distance by at most t^*.

So the overall change in potential is at most $2(r^* - 1 + t^* - (r - r^*)) = 4r^* + 2t^* - 2r - 2$.

→ In heuristic algorithm, first element moves r-1 positions
→ In optimum algorithm, first element moves r*-1 positions. t* swaps happen.
→ change because of r*-1 positions movement in optimum algorithm
→ Total change = r*-1 - (r-1 - r*-1) + t* = 2r* - r + t* - 1
→ Potential diff = 4r* + 2t* - 2r – 2

(e) Now complete the amortize analysis using the above potential functio the cost of the above mentioned heuristic algorithm is at most $4C_{opt}$.

Answer: The amortized cost of operation s_t is actual cost + potential difference, $(2r - 1) + (4r^* + 2t^* - 2r - 2) \le 4(r^* + t^*) = 4 \times$ Actual cost of optimal algorithm for Hence the total cost of all the operations by the heuristic algorithm is at most $4C_{opt}$.

Note, what you have just seen is called the *competitive analysis* in the domain of onli the above question you have finally shown that your online algorithm is 4-competitive. online algorithms (and competitive analysis) in the last two lectures of this course.

→ Calculate amortized cost to be less than $4*C_{opt}$.

Partition Array of integers with sum S to k buckets of values S/k each

```
dp[0] = 0
target = S/k
# Iterate over all subsets/masks
for mask in range((1 << N)):

    # If current mask is invalid,
    # continue
    if (dp[mask] == -1):
        continue

    # Iterate over all array elements
    for i in range(N):

        # Check if the current element
        # can be added to the current subset/mask
        #     - mask & (1 << i) == 0 ensures that the bit at i = 0 in the mask
        #     - dp[mask] + arr[i] <= target ensures that the value can be added
        #       into the unfilled partition
        if ((mask & (1 << i) == 0) and dp[mask] + arr[i] <= target):
            # Transition
            dp[mask | (1 << i)] = ((dp[mask] + arr[i]) % target)

if (dp[(1 << N) - 1] == 0):
    return True
else:
    return False
```

Greedy Algorithm Strategy

1. **Describe the algorithm and the time complexity**.
2. **Explain its correctness and greedy choice property**: Suppose there is some optimal solution S with <some attribute>. We replace <this attribute> with the <greedy attribute>. Let S' be the solution obtained by taking S and replacing <attribute> with <greedy attribute>. We claim that <improve result without extra costs, i.e. it doesn't hurt to take this choice>. Explain the claim. Hence, S' is a valid solution and is also an optimal solution.
3. **Explain optimal substructure property**: Let P be the original problem with optimal solution S. Let P' be the subproblem finding the solution for <describe the problem again but with less of some parameters based on the greedy solution>. Explain that S' is also optimal and S = S' + <parameter>

3-SAT reduce to EQ-GFS (i.e. Prove EQ-GFS is NP Complete)

Question 4: EQ-GF2

(Taken from AY21/22 S1 Assignment)

The EQ-GF2 problem is the following: given a system of polynomial equations over integers modulo 2, determine if there is an assignment that satisfies all the equations simultaneously. For example, $\{x + y = 0, xy + y = 1\}$ is a NO-instance of EQ-GF2, while $\{x + y = 1, xy + y = 1\}$ is a YES-instance satisfied by $x = 0, y = 1$.

Show that EQ-GF2 is NP-complete. Consider a reduction from 3-SAT.

Note: Integers modulo 2 simply mean that only 0 and 1 are the possible values that each variable can take an evaluate to. This means that if we have equations x + y, and x +1, y = 1, x + y will evaluate to 0 (because 1 + 1 (mod 2)). Furthermore, while the example above suggests only the usage of x and y, you are actually allowed to have more variables. Therefore similarly if we have x + y + z and all the values are set to 1, it is equivalent to 1 + 1 + 1 (mod 2) = 1

Tip when reducing from 3-SAT: Try to think how to model components of 3-SAT in the problem you are reducing to. 3-SAT has the notion of (1) 'literal' -- which can be *exclusively* true or false, and (2) 'clause' -- which contains 3 literals and *at least one* of them must be set to true. Then, find a way to connect these together.

Answer

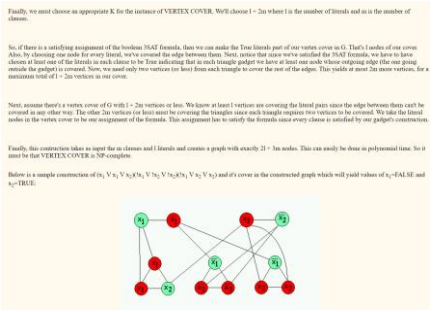
The idea behind the reduction is to observe that $x \vee y \vee z$ is true if and only if $\neg x \wedge \neg y \wedge \neg z$ is false. Now, if we think of 1 (mod 2) (or odd integers) as representing true and 0 (mod 2) (or even integers) as representing false, then conjunction naturally corresponds to multiplication: product is odd iff every multiplicand is odd. In this encoding, negation is equivalent to adding 1, as this shifts odds to evens and evens to odds. Hence, $x \vee y \vee z$ can be encoded as:

$$(1 + x)(1 + y)(1 + z) = 0$$

over the integers mod 2. Similarly, $\neg x \vee y \vee z$ becomes $x(1 + y)(1 + z) = 0$, etc.

Thus, the reduction is to take every clause of the 3-SAT formula and produce the corresponding polynomial over integers mod 2 in the manner described above. The reduction is in polynomial time, because each 3-SAT clause is of size at most 3, so only a constant amount of time is spent on each clause, yielding a linear time reduction. The correctness is due to the first paragraph.

3-SAT Reduce to Vertex Cover



→ Arguing that yes(3-sat) → yes(vertex-cover): If there is a yes instance to 3-sat, then there is a satisfying assignment such that the entire CNF is true. Which means that there is some literal inside each clause that evaluates to true. And if that literal is true, it has covered the edges concerning its edge between its inverse truth value and all the edges towards its assignment in the clauses. Now, we can just take the other 2 literals inside each clause and shade it as part of the vertex cover. We have now covered all edges

→ Arguing that yes(vertex-cover) → yes(3-sat): If we have a yes instance to vertex cover, then we have some set of nodes that cover all edges in <= l + 2m nodes. We claim that every single clause will have some literal that results the clause to be true. Assume we did not, then there is some clause with 3 nodes used to cover the edges in the answer, which will result in the vertex cover having > l + 2m nodes, a contradiction. So, now that all edges are covered, we can then take the shaded literal in the dumbbell to get a satisfying 3sat solution.

Vertex Cover <=p Dominating Set

Question 6: DOMINATING-SET

(Taken from AY21/22 S1 Assignment)

A dominating set of graph $G = (V, E)$ is defined to be a subset of vertices $S \subseteq V$ such that every vertex of G is either in S or adjacent to a vertex in S. The dominating set decision problem is the problem of deciding, given a graph G and an integer k, whether G contains a dominating set of size at most k or not. Show that this problem does not have a polynomial-time algorithm, unless P = NP.

(Hint in lighter font: Consider a reduction from Vertex Cover.)

Answer

We give a polynomial time reduction from Vertex Cover to Dominating Set.

Given an instance of vertex cover (a graph G and a number k), construct an instance of Dominating Set (a graph G' and a number k') as follows. Let $k' = k$. For G', start with G and add to it a new vertex labeled ab for every edge (a, b) in the graph G. Also, put in new edges (ab, a) and (ab, b) in G' for each edge (a, b) in G. (By the way, G' is said to be the *line graph* of G.)

The reduction is in polynomial time because it needs to add one new vertex and two new edges for every edge in the old graph. So, in fact, this reduction runs in time linear in the size of the input to Vertex Cover.

Suppose (G, k) is a YES-instance of Vertex Cover. Then, if S is a subset of size at most k that is a vertex cover, then, we claim that S is also a dominating set for G'. This is easy to check as S is clearly a dominating set for any vertex in G. Also, any new vertex in G' (corresponding to an edge (a, b) in G) is also adjacent to S, because S must contain either a or b as it is a vertex cover. Hence, (G', k) is also a YES-instance of Dominating Set.

Next, suppose (G', k) is a YES-instance of Dominating Set. Let S be a witnessing dominating set of size at most k in G'. Note that S can contain both vertices from G as well as the new vertices. We first: (i) modify S to S' so that S' is a subset of the original nodes, and (ii) argue that S' is a vertex cover of G. For (i), if S contains a node ab, then we replace it with either a or b (the choice doesn't matter) in S'. For (ii), take any edge (u, v) in G. Now, S must have contained either uv or u or v, because S was a dominating set in G'. In any case, S' will contain either u or v, and hence, S' must cover (u, v) in G.

- 3 points for giving reduction in right direction
- 1 point for arguing poly time
- 2 points for showing YES instance of Vertex Cover (or other NP-hard problem) map to YES instance of Dominating Set
- 2 points for showing the converse.
- 0 points for showing reduction in opposite direction.

Max-Clique <=p Bipartite Graph Family

Question 7: Bipartite Graph Family

(Taken from AY20/21 S2 Assignment)

Note: I have never attempted this question himself. So he might not be able to provide much hints about this or provide an explanation of the answer key (until he is asked to). But I am keeping it here in case you really have tried all the questions above and you'd like more practice, so I don't want to deprive you of that.

Question 1 [20 marks]: An undirected bipartite graph $G = (L, R, E)$ has two disjoint set of vertices L and R, and for every edge $e \in E$, one endpoint must be in L and another in R. We call a pair of distinct vertices $u, v \in L$ *siblings* if there exists a vertex $r \in R$ such that both the edges (u, r) and (v, r) are present (in E). A subset $F \subseteq L$ is said to be a *family* if for all the distinct $u, v \in F$, u and v are siblings.

Now, consider the following problem: Given an undirected bipartite graph $G = (L, R, E)$ and a positive integer k, decide whether there is a family $F \subseteq L$ of size at least k.

Show that the above problem is NP-complete. [You may show a reduction from any of the NP-complete problems introduced in the lectures/tutorials, more specifically, Circuit Satisfiability, CNF-SAT, 3-SAT, Vertex Cover, Independent Set, Max-Clique, Hamiltonian Cycle, Traveling Sales Person Problem.]

[Hint: Try a reduction from the Max-Clique problem.]

Max-Clique in the hint is the same problem as in the tutorial. That is the decision problem that decides whether there is a clique of size at least k in G?

Answer

To show that the problem is NP-Complete, we first show that it is in NP. We can make a simple verifier V such that $V(\langle G, k \rangle, F)$ verifies whether the set F is a family of bipartite graph G satisfying $|F| \ge k$.

- Parse G as bipartite graph (L, R, E).
- If $F \subset L$ continue, else reject.
- if $|F| \ge k$ continue, else reject.
- For each distinct $u, v \in F$, check whether there exists $r \in R$ such that $(u, r) \in E$ and $(v, r) \in E$. If such r exists for all distinct u, v, accept, otherwise reject.

The verifier is simply a direct interpretation of the problem definition.

To show that the problem is NP-Hard, we can do a reduction from Max-Clique problem. The reduction function takes as input (G, K) from Max-Clique (is there a clique of size at least K in G?), where $G = (V, E)$ is a graph and $K \ge 0$ is an integer, and outputs a bipartite graph $G' = (L, R, E')$ and integer k, as follows.

- $L = V$ (i.e., add a vertex in L for each vertex in G).
- $R = E$ (i.e., add a vertex in R for each edge in G).
- E' is the set of all (v, e) such that $v \in V$, $e \in E$, and v is an endpoint of the edge e, (that is, each vertex v is connected to the edges it is attached to in G).
- $k = K$

To check if this reduction holds, first, assume G has a clique C of size at least K. Then, $F = C$ is a family of size at least k, since all pairs of vertices in C shares an edge, and thus each pair has a vertex in R representing said edge. Conversely, assume G' has a family F' of size at least k, then $C' = F'$ is a clique of size at least K. Since both holds, the problem is NP-Hard.

Since the problem is shown to be both in NP and NP-Hard, it is NP-Complete.

Fibonacci Amortization

Q4. (Continued …)

(b) Suppose Alice insists Bob to maintain a dynamic table (that supports both insertion and deletion) in such a way its size must always be a Fibonacci number. She insists on the following variant of the rebuilding strategy. Let F_k denote the k-th Fibonacci number. Suppose the current table size is F_k.
After an insertion, if the number of items in the table is F_{k+1}, allocate a new table of size F_{k+1}, move everything into the new table, and then free the old table.
After a deletion, if the number of items in the table is F_{k-2}, we allocate a new hash table of size F_{k-1}, move everything into the new table, and then free the old table.
Use either Potential method or Accounting method to show that for any sequence of insertions and deletions, the amortized cost per operation is still O(1). (If you use Potential method clearly state your potential function. If you use Accounting method clearly state your charging scheme.)

Soln: Suppose the current table size is F_k, then there must be at most F_{k-1}elements present in the table.
Consider the following charging scheme for insertion: Charge $4 for each insertion, use $1 for insertion and remaining $3 put in the bank. Between the creation of table of size F_{k+1} and F_{k+2} there must be at least $F_k - F_{k-1} = F_{k-2}$ elements being inserted. So the bank balance at the time of creating table of size F_{k+2} must be $3F_{k-2} > F_{k-3} + 2F_{k-2} = F_{k-1} + F_{k-2} = F_k$. So we can move all the F_k elements to the new table using the bank balance (free of cost). Hence the amortization cost of insertion is O(1).
For the deletion consider the following charging scheme: Charge $3 for each deletion, use $1 for deletion and remaining $2 put in the bank. Between the creation of table of size F_{k-1} and F_{k-2} there must be at least $F_{k-3} - F_{k-4} = F_{k-5}$ elements being deleted. So the bank balance at the time of creating table of size F_{k-2} must be $2F_{k-5} > F_{k-6} + F_{k-5} = F_{k-4}$. So we can move all the F_{k-4} elements to the new table using the bank balance (free of cost). Hence the amortization cost of deletion is O(1).
Note, at any point bank balance is some constant times the number of elements present in the table, and hence is non-negative.

Set Union Problem (Amortized O(logn))

Q4. (10+10 = 20 points)

(a) In the Set Union problem we have n elements, that each are initially in n singleton sets, and we want to support the following operations:
- **Union(A,B):** Merge the two sets A and B into one new set C = A∪ B destroying the old sets.
- **SameSet(x, y):** Return true, if x and y are in the same set, and false otherwise.

We implement it in the following way. Initially, give each set a distinct color. When merging two sets, recolor the smaller (in size) one with the color of the larger one (break ties arbitrarily). Note, to recolor a set you have to recolor all the elements in that set.
To answer **SameSet** queries, check if the two elements have the same color. (Assume that you can check the color an element in O(1) time, and to recolor an element you also need O(1) time. Further assume that you can know the size of a set in O(1) time.)

Use Aggregate method to show that the amortized cost is O(log n) for **Union**. That means, show that any sequence of m union operations takes time $O(m \log n)$. (Note, we start with n singleton sets.)

Soln: Initially we have n singleton set. So any sequence of m union operations can involve at most $2m$ many elements. By our implementation, cost of a union operation is equal to the number of elements recolored during that operation. Hence total cost of m operations is at most twice the total number of recoloring happens. Now count the total number of recoloring by element wise. Observe that each element can be recolored at most $\log n$ times. This is because since we are recoloring smaller set, an element is recolored k times means it was part of smaller set k times, and each time the size doubles. More specifically, if we consider all these k unions where that element is part of the smaller set, and let $s_1, \ldots, s_k$ be the sizes of those smaller sets. Clearly $s_i \geq 2s_{i-1}$, and so $k \leq \log n$.Thus total number of recoloring is bounded by $O(m \log n)$.

Given some string, how many distinct subsequences are there?

Q2. (20 points) Distinct Subsequences

Given an n-length string of English letters, consider the problem of counting the number of distinct subsequences (not necessarily contiguous) appearing in it. For example, the string *aab* contains six distinct subsequences (empty string, *a*, *b*, *aa*, *ab*, *aab*), the string *aba* contains seven (empty string, *a*, *b*, *aa*, *ab*, *ba*, *aba*), while for the string *abc*, all the eight subsequences are distinct.

(a) Let $N[i]$ denote the number of distinct subsequences among the first i characters of a string x. Write a recurrence relation for $N[i]$ in terms of $N[j]$ with $j < i$, and justify.

Hint: Let $prev(i)$ be the largest j in $\{1,2,\ldots,i-1\}$ such that $x_j = x_i$. if there exists one. Note that for every subsequence whose last element is $x_{prev(i)}$, there is another identical subsequence that ends at x_i.

ANSWER: Denote the string as x. Let $prev(i)$ be the largest $j < i$ such that $x_j = x_i$ and undefined otherwise. Then $N[0] = 1$, and for $i > 0$:

$$N[i] = \begin{cases} 2 \cdot N[i-1], & \text{if } prev(i) \text{ is undefined} \\ 2 \cdot N[i-1] - N[prev(i)-1], & \text{otherwise} \end{cases}$$

Proof: Let S_i be the set of distinct subsequences among the first i characters of the string.

If x_i hasn't occurred before (i.e., $prev(i)$ is undefined), then S_{i-1} and $\{s \circ x_i : s \in S_{i-1}\}$ are disjoint, so $N[i] = 2 \cdot N[i-1]$.

Otherwise, the items in the overlap $S_{i-1} \cap \{s \circ x_i : s \in S_{i-1}\}$ are exactly the strings in $S_{prev(i)-1}$ with x_i appended. So, $N[i] = 2 \cdot N[i-1] - N[prev(i)-1]$.

Subset Sum <=p Partition

(b) Consider the following two problems:

- PARTITION: Given a set of n non-negative integers $\{a_1, \ldots, a_n\}$, decide if there is a subset $P \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$..
- SUBSETSUM: Given a set of n non-negative integers $\{a_1, \ldots, a_n\}$ and an integer T, decide if there is a subset $P \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in P} a_i = T$.

Reduce SUBSETSUM to PARTITION.

ANSWER: Given an instance of SUBSETSUM $\{a_1, \ldots, a_n\}$ and T, let $S = a_1 + a_2 + \cdots + a_n$. Assume S is not 0 or T, and T is not zero, as otherwise the problem is trivial.

Consider the solution to PARTITION on the input $\{a_1, \ldots, a_n, S + T, 2S - T\}$. The total sum of all the elements in the input is $4S$.

If the SUBSETSUM instance is a YES-instance, i.e., there is a subset P such that $\sum_{i \in P} a_i = T$, then the PARTITION instance is also a YES-instance because $\sum_{i \in P} a_i + 2S - T = 2S$.

Now, suppose the PARTITION instance is a YES-instance. Note that by assumption, $S + T, 2S - T \neq 2S$ and $(S + T) + (2S - T) > 2S$. So, it must be that $2S - T$ is in one partition and $S + T$ is in the other. Then, the rest of the elements in the partition containing $2S - T$ must sum up to T, forming a solution to the SUBSETSUM instance.

Factor <=p Euler Totient

Q4. (20 points) Reductions

(a) Consider the following two problems:

- FACTOR: Given a positive integer n, the product of two distinct primes p and q, find p and q.
- EULERTOTIENT: Given a positive integer n, the product of two distinct primes p and q, find $\phi(n) = (p-1)(q-1)$.

Reduce FACTOR to EULERTOTIENT.

ANSWER:
Suppose we have an algorithm to compute $\phi(n) = (p-1)(q-1)$. Then, we can compute $b = n - \phi(n) = p + q - 1$. So, $q = b + 1 - p$. Therefore, if we solve the quadratic equation $p(b + 1 - p) = n$ using the quadratic formula, we can recover p and then q, solving FACTOR.

Decimate Amortized Problem

Question 1 [15 marks]: Consider a standard (FIFO) queue that supports the following operations:

- PUSH(x): Add item x at the end of the queue.
- PULL(): Remove and return the first item present in the queue.
- SIZE(): Return the number of elements present in the queue.

We can easily implement such a queue using a doubly-linked list so that each of the above three operations takes $O(1)$ worst-case time. (No need to show this. You can assume such an implementation.) Now, suppose we are asked to consider the following new operation:

- DECIMATE(): Remove every tenth element from the queue starting from the beginning.

```
DECIMATE():
  n ← SIZE()
  for i ← 0 to n − 1
      if i mod 10 = 0
          PULL()   {result discarded}
      else
          PUSH(PULL())
```

Figure 1: Pseudocode of the DECIMATE operation (assuming starting index to be 0)

The above DECIMATE operation takes $O(n)$ time in the worst-case (where n is the number of items present in the queue). Show that in any intermixed sequence of PUSH, PULL and DECIMATE operations, the amortized cost of each of them is $O(1)$. (Use any one of the three methods we have learned in the module, i.e., aggregate method, accounting method, or potential method.)

Solution: Suppose at any point of time the number of elements present in the queue is ℓ. Then the worst-case (actual) running time of PUSH(x), PULL(), SIZE() and DECIMATE() are c_1, c_2, c_3 and $c_4\ell$ respectively, for some constant $c_1, c_2, c_3, c_4 > 0$. Now, we use the potential method to establish the desired bound on the amortized costs of the above four operations.

Let us consider the following potential function: $\phi = 10c_4 \times$ the number of elements present in the queue. Clearly, in the beginning $\phi = 0$, and at any point of time $\phi \geq 0$. So ϕ is a valid potential function. Recall, in the potential method, the amortized cost of an operation is equal to the sum of its actual cost and the potential difference after the application of that particular operation.

Next, consider the amortized cost of each of the four operations:

- PUSH(x): After this operation, the potential difference is $10c_4$, and thus the amortized cost is $c_1 + 10c_4$.
- PULL(): After this operation, the potential difference is $-10c_4$, and thus the amortized cost is $c_2 - 10c_4$.
- SIZE(): After this operation, the potential difference is 0, and thus the amortized cost is c_3.
- DECIMATE(): Let the number of elements present in the queue is ℓ. After this operation, the potential difference is $-10c_4\lfloor \frac{\ell}{10} \rfloor$, and thus the amortized cost is $c_4\ell - 10c_4\lfloor \frac{\ell}{10} \rfloor \leq 10c_4$.

So, we conclude that the amortized cost of each of the four operations is $O(1)$.

→ Can also use accounting method, by assigning cost of 20 – 1 for cost of insert, 19 in the bank.
→ For pull(), use 1 from the bank.
→ For halve, for every 10th element, use 1 from the bank to pull() itself, and the other 18 to pull() and push() the other 9 elements that are added back into the queue.

(r,l) separated subsets

Question 2 [25 marks]: For any set S of integers and two non-negative integers r, ℓ, we call a subset $S' \subseteq S$ an (r,ℓ)-*spaced subset* if the sum of all the integers in S' is r and for every two distinct integers $a, b \in S'$, $|a - b| \geq \ell$. Describe an algorithm that given two non-negative integers r and ℓ, and a set S of n distinct non-negative integers, counts the number of possible (r,ℓ)-spaced subsets of S in $O(rn \log n)$ time. (Assume, a single machine word is large enough to hold any integer computed during your algorithm.)
[Significant partial credits will be awarded if your algorithm runs in time $O(rn^2)$.]

Solution: First, sort the integers in S into an array $A[0, \cdots, n-1]$ (of size n). Then for $i \in \{0, 1, \cdots, n\}$ and $j \in \{0, 1, \cdots, r\}$, define $x(i,j)$ as the number of (j,ℓ)-separated subsets of $A[i], A[i+1], \cdots, A[n-1]$. Note, the number of possible (r,ℓ)-spaced subsets of S is $x(0,r)$.

As base case, $x(n,0) = 1$ (corresponds to the empty subset S'), and $x(n,j) = 0$ for all $j > 0$. To get the optimal substructure property, sum the number of (j,ℓ)-separated subsets using $A[i]$ with the ones that do not use $A[i]$. So we do the following:

If $A[i] \leq j$ is used, then

- No integer in $A[i], A[i+1], \cdots, A[n-1]$ smaller than $A[i] + \ell$ can be used.
- Let $f(i)$ be the smallest index greater than i such that $A[f(i)] - A[i] \geq \ell$.
- Use the value $x(f(i), j - A[i])$.

Else, $A[i]$ is not used and we can use the value $x(i+1, j)$.
The recursive formula for the $x(i,j)$ is the following:

$$x(i,j) = \sum \left(\begin{cases} x(f(i), j - A[i]) & \text{if } A[i] \leq j \\ x(i+1, j) & \text{always} \end{cases} \right).$$

The number of subproblems is $(n+1) \times (r+1) = O(rn)$. Time taken per subproblem is $O(\log n)$ (since we can find $f(i)$ by performing a binary search over the sorted array A). Sorting of integers in S (to form A) can also be done in $O(n \log n)$ time (by using merge sort). So the total time taken is $O(rn \log n)$.

→ Find the number of ways to from a subset of the array where the sum is r and each value is >= l from each other.
→ Is basically like knapsack problem, except that you need to make sure the item u took from is >=l distance from you. So we sort the array first, then perform binary search for the first element that is >= l from you
→ It is like knapsack because its like "take or don't take", if the sum is less than n, we keep adding. And instead of looking from the max of either take or don't take, we sum all the possible ways together.

Vertex Cover <=p Hitting Set (Prove Hitting Set is NP Complete)

Question 3 [15 marks]: A set H is said to be a *hitting set* for a family of sets $\{S_1, S_2, \cdots, S_n\}$ if and only if for all $1 \leq i \leq n$, $H \cap S_i \neq \phi$ (i.e., H has a non-empty intersection with all the sets S_i).
Consider the following hitting set problem: Given a family of finite integer sets $\{S_1, S_2, \cdots, S_n\}$ and a positive integer K, decide whether there exists a hitting set of size at most K. Show that the hitting set problem is NP-complete.
(You may show a reduction from any of the NP-complete problems introduced in the lectures/ tutorials/ assignments/ practice set, including Circuit Satisfiability, CNF-SAT, 3-SAT, Vertex Cover, Independent Set, Max-Clique, Hamiltonian Cycle, Traveling Sales Person Problem.)

Solution: The hitting set problem is clearly in NP because a hitting set H of size at most K can act as a valid certificate. Verifying the certificate also can easily be done in polynomial time by checking whether it is of size at most K and also has a non-empty intersection with all the input sets.
Next, we can show a polynomial-time reduction from the vertex cover problem. Take an instance (G,k) of the vertex cover problem. Then for each edge $e = (u,v)$ of the graph G, create a set $S_e = \{u,v\}$. (Consider an arbitrary integer labeling of the vertices.) Then consider the family of sets $\mathcal{F} = \{S_e | e \text{ is an edge in } G\}$ and set $K = k$. This constitute an instance of the hitting set problem. It is easy to see that (G,k) is a yes-instance of the vertex cover if and only if $(\mathcal{F}, k)$ is a yes-instance of the hitting set. Hence, we deduce that the hitting set problem is NP-complete.