COSC 3320 – Algorithms & Data Structures

Module 1: Intro to Algorithms

Chapter 1: Introduction

Find the Celebrity Problem

A celebrity among a group of n people is a person who knows no one but is known by everyone else.

Goal: Identify a celebrity by only asking questions to people of the form: "Do you know this person?"

For 1 person: Does A know B?

$n \cdot (n-1) \longrightarrow n(n-1) = n^2 - n = \Theta(n^2)$

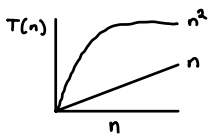
IMPORTANT!! "polynomial in" input size → time complexity

$\Theta(n^2)$ - good, but can be better

$\Theta(n)$ - better for now

time complexity graph

- "scaling" with respect to input size

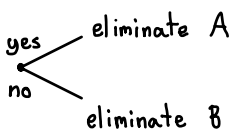


Decrease & Conquer Method

- Recursive

ex:

Does A know B?



$n \to n-1 \to$ repeat the step

- $P(n) \to P(n') \to \ldots \to P(1)$
  where $n' < n$
  – reduce the problem into smaller chunks

Once we have: $n \to n-1 \to \ldots \to 1$

$(n-1)$ • First time, "Do you know this person?"

$(n-1)$ • Once there's one last person, ask if they know everyone else

$(n-1)$ • Ask everyone if they know the last person

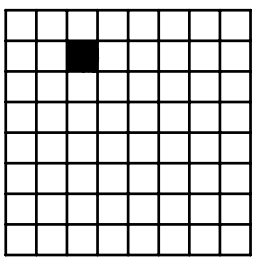
Finally, $(n-1)+(n-1)+(n-1) = 3n-3 \to \Theta(n)$

An even better algorithm is:

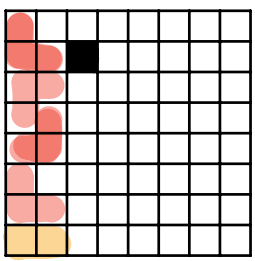
- $3n - 4$
- $3n - \lfloor \log(n) \rfloor - 3$

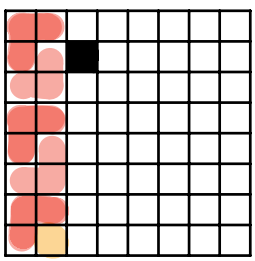
Tiling Problem

Given a $2^n \times 2^n$ board with one missing square, use L-shape to fill the grid



- Backtracking

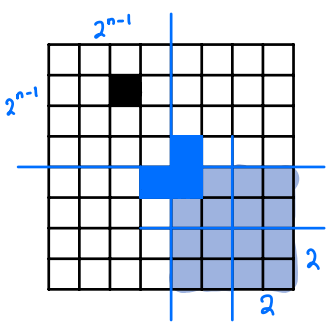




↑ not going to fit here, so backtrack

- good method, but may take a long time to find a solution
- always find a solution

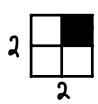
- Use recursion to solve this problem (Divide and Conquer)



Note: $4 \cdot (2^{n-1} \times 2^{n-1}) = 2^n$

- the problem can be divided into 4 subproblem

If we repeat this n-1 times, we arrive at 2×2



easier to solve now

where the base case is $2^n \times 2^n$.

Chapter 2.1 - 2.2 : Problem Solving

Is n prime?

1. Understand < input → n, positive integer
output → yes/no

- a natural number greater than 1 is prime if it is divisible only by itself and 1

2. Problem Solving

- Divide/Decrease & Conquer
- Dynamic Programming
- Greedy

Algorithm - Pseudocode

3. Analysis < characteristic (proof by induction)
performance (time/space complexity)

Is the algorithm efficient? "polynomial in" input size

4. Coding

PseudoCode

```
prime(n)
    for i = 2 to n-1
        if i divides n
            return "no"
    return "yes"
```

← $O(n-1) = O(n)$ time complexity

Is the code efficient?

No, the input size is $\log_2(n)$ so:

$O(n)$

$n = 2^{\text{input size}} = 2^{\log_2(n)}$

the time complexity is exponential.

$i \times \frac{n}{i} = n \rightarrow i = \frac{n}{i}$

$i^2 = n$

$i = \sqrt{n}$

```
prime(n)
    for i = 2 to ⌊√n⌋
        if i divides n
            return "no"
    return "yes"
```

$\sqrt{n}$ is closer to 1

Is the code efficient?

Kind of, but not really.

$n = 2^b$ where $b = \log_2(n)$

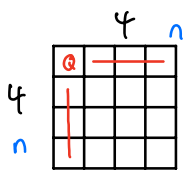
$\sqrt{n} = \sqrt{2^b} = (\sqrt{2})^b = (1.414)^b$

$(1.414)^b < 2^b$

$(1.464)^b$ is still exponential, not polynomial.

Prime algorithm should aim for $O\left(\frac{n}{\log(n)}\right)$?

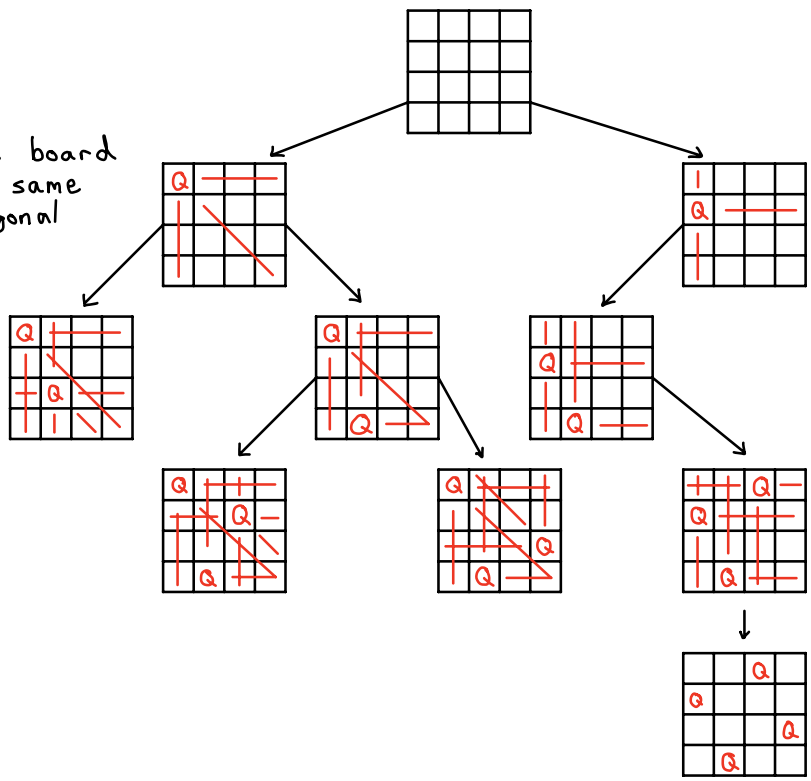
n-Queens problem



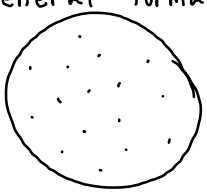
Place queens on the board without being on the same row, column, or diagonal

$\binom{n^2}{n} > \left(\frac{n^2}{n}\right)^n = n^n$

- use backtrack to solve



General format:



Find a specific dot.

You can check each individual dot, but not very efficient.

(aka. Brute force method)

Naive Algorithm:

- The Integer Square Root Problem

Find $\sqrt{n}$

1 . . . . . n

Pseudo Code:

```
Is SQRT(n):
    for i = 1 to n
        if i^2 ≤ n and (i+1)^2 > n
            return i
```

$O(\sqrt{n})$
- not efficient

Binary Search

- Go to the mid-point
- Choose right or left
- Reduce by a factor of 2
- Find : $x^2 \leq n$ and $(x+1)^2 \geq n$

$O(\sqrt{n})$ $\quad \sqrt{n} = 2^{\frac{1}{2}\log_2(n)} = (\sqrt{2})^{\log_2(n)}$ ← input size

search

n

$\frac{n}{2^1}$

$\frac{n}{2^2}$

$\vdots$

$\frac{n}{2^{\log_2(n)}} = 1$

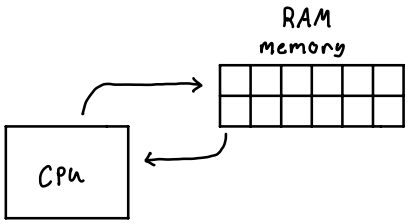
Pseudo Code

```
search(low, high, n):
    mid = ⌊(low + high)/2⌋
    termination/base case {
        if mid^2 ≤ n and (mid+1)^2 ≥ n
            return mid
    recursive {
        else if mid^2 < n
            search(mid, high, n)
        else
            search(mid, low, n)
```

$O(\sqrt{n}) > O(\log_2(n))$

RAM Model

Sequenctial computation



1. CPU
   one step takes one operation
2. Read/Write
   one memory CPU per unit step

Complexity = # of operations

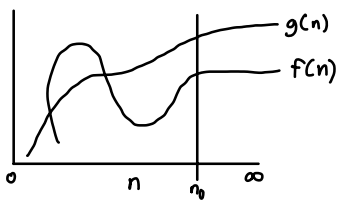
Chapter 2.5 : Big-O

Big-O Notation:

Definition : Let $f(n)$ and $g(n)$ be positive functions.

$f(n)$ is Big-O of $g(n)$, or
$f(n) = O(g(n))$

- Upper bound is Big-O



If there exist positive constants c and n_0 , for all $n \geq n_0$
$f(n) \leq c\, g(n)$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$, then $f(n) = O(g(n))$

n	O(1)	O(log n)	O(n)	O(n log n)	O(n²)	O(2ⁿ)	O(n!)

Big-Ω Notation:

Fact : $g(n) = O(f(n))$ is the same as $f(n) = \Omega(g(n))$

Definition : Let $f(n)$ and $g(n)$ be positive functions.

$f(n)$ is Big-Ω of $g(n)$
$f(n) = \Omega(g(n))$

- Lower bound is Big-Ω

If there exist positive constants c and n_0, for all $n \geq n_0$
$f(n) \geq c\, g(n)$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$, then $f(n) = \Omega(g(n))$

Big - Θ Notation

Definition: Let $f(n)$ and $g(n)$ be positive functions.

$f(n)$ is Big - Θ of $g(n)$
$f(n) = \Theta(g(n))$

If there exist positive constants c_1, c_2, and n_0, for all $n \geq n_0$
$c_1 g(n) \leq f(n) \leq c_2 g(n)$

If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n) = \Theta(g(n))$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, then $f(n) = \Theta(g(n))$

ex:
$f(n) = n\log(n)$
$g(n) = n^2$

$$\lim_{n\to\infty} \frac{n\log(n)}{n^2} = \frac{\frac{d}{dn}\log(n)}{\frac{d}{dn}n} = \lim_{n\to\infty} \frac{1/n}{1} = \lim_{n\to\infty} \frac{1}{n} = 0$$

so, $f(n) = O(g(n))$ and
$f(n) = \Theta(g(n))$

little - o Notation

$f(n) = o(g(n))$ $\quad f(n) < cg(n)$ for all $c > 0$ $\quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

little - ω Notation

$f(n) = \omega(g(n))$ $\quad f(n) > cg(n)$ for all $c > 0$ $\quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Theorem:

Suppose $\log(f(n)) \leq c \log(g(n))$ for some $0 \leq c \leq 1$

$f(n) = O(g(n))$

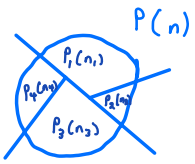
If $\log(f(n)) = o(\log(g(n)))$, then $f(n) = O(g(n))$

Runtime Table

Runtime	Terminology
$O(n)$	Linear
$O(n^2)$	Quadratic
$O(n^c)$, $c > 1$	Polynomial
$O(c^n)$, $c > 1$	Exponential
$O(\log(n))$	Logarithmic

Chapter 3 : Mathematical Induction

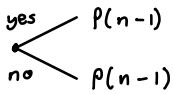
Celebrity Problem

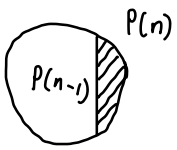


- Decrease and Conquer algorithm

P(n) : celebrity in n people

A knows B ?





P(1) : ASSUME n = 1 is the celebrity

$P(n) \rightarrow P(n-1) \rightarrow P(n-2) \rightarrow \cdots \rightarrow P(1)$ ✓

Proof by Induction

Base Case: n = 1 (vacuosly true)

Induction Step:

Induction hypothesis : Assume that the strategy is correct for k people.
We will show that the strategy is correct for k+1 people.

Applying the strategy for k+1, we eliminate 1 person. It is clear that the eliminated person cannot be a celebrity. Thus, if there is a celebrity, then it has to be among the remaining k people.

Hence the strategy is correct for k+1 people. □

Tiling Problem

- Divide and Conquer algorithm
- Number of tiles needed to cover a $2^n \times 2^n$ grid with a missing tile

$$\frac{(2^n)(2^n)-1}{3} = \frac{4^n-1}{3}$$

- Recurrence Technique
  - reduce to 4 subproblems of size $2^{n-1} \times 2^{n-1}$

$T(n)$ in terms of $T(n-1)$

$T(n) = 4T(n-1) + 1$, where $T(n-1)$ is a $2^{n-1} \times 2^{n-1}$ grid (with 1 missing tile)

$+1$ is the 1 tile placed in the middle

- What about the Base Case?

Only 1 L-shaped tile is needed to fill a 2×2 grid, so $T(1) = 1$

Theorem: The solution to the recurrence $T(n) = 4T(n-1) + 1$, with $T(1) = 1$ is $T(n) = \frac{4^n-1}{3}$

Proof by Induction

Base Case: $n = 1$

$T(1) = \frac{4^1-1}{3} = 1$, which is true.

Induction Step:

Inductive hypothesis: Assume that the statement is true for k.
We will prove that it holds for k+1.

By the recurrence and the induction hypothesis,

$T(k) = \frac{4^k-1}{3}$

We want to find $T(k+1)$
$\frac{4^{k+1}-1}{3}$

$$\begin{aligned} T(k+1) &= 4T(k+1-1) + 1 \\ &= 4T(k) + 1 \\ &= 4\left(\frac{4^k-1}{3}\right) + 1 \\ &= \frac{4^{k+1}-4}{3} + \frac{3}{3} \\ &= \frac{4^{k+1}-4+3}{3} \\ &= \frac{4^{k+1}-1}{3} \end{aligned}$$

□

Chapter 4: Recursion

GCD

- greatest common divisor
- gcd (a, b)
- Euclid's algorithm

$a \geq 0$
$b \geq 0$
$a \geq b$
$(a \bmod b) < b$

$$\gcd(a, b) = \gcd(b, a \bmod b)$$
$$= \gcd(a \bmod b, b \bmod (a \bmod b))$$

PsendoCode
gcd (a, b):

termination call { if b = 0
return a
recursive call { else
return gcd (b, a mod b)

$b \xrightarrow{2} \frac{b}{2} \xrightarrow{2} \frac{b}{4} \to \dots \to 1 = O(\log(b))$

Case 1: $a \bmod b < \frac{b}{2}$

Case 2: $a \bmod b \geq \frac{b}{2} \xrightarrow{1} < \frac{b}{2}$

$O(\log(b)) \longleftarrow 2\lceil \log(b) \rceil + 1$

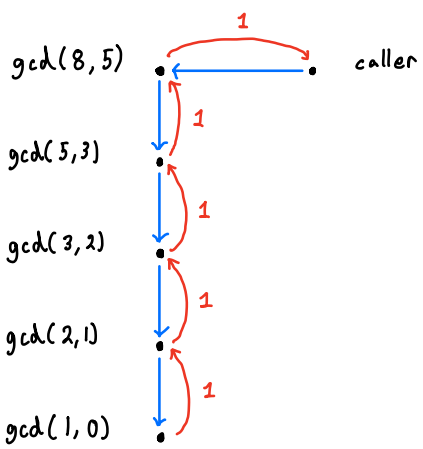
Recursion flow diagram:

a = 8 and b = 5

Recursive Call

Return Value

- Also called Recursion Tree



- termination call = "base case"

Search for the MAX element

Given an array $S = [n_0, n_1, \ldots n \text{ elements}]$, find max value in S.

- Sequential search
  - compares each element with the current max
  - sequential order (1^{st}, 2^{nd}, 3^{rd}, ... n^{th})

Pseudo Code

MAX (S):

```
max = S[1]
for i = 2 to n
    if max < S[i]          ⟶  n comparisons
        max = S[i]
return max
```

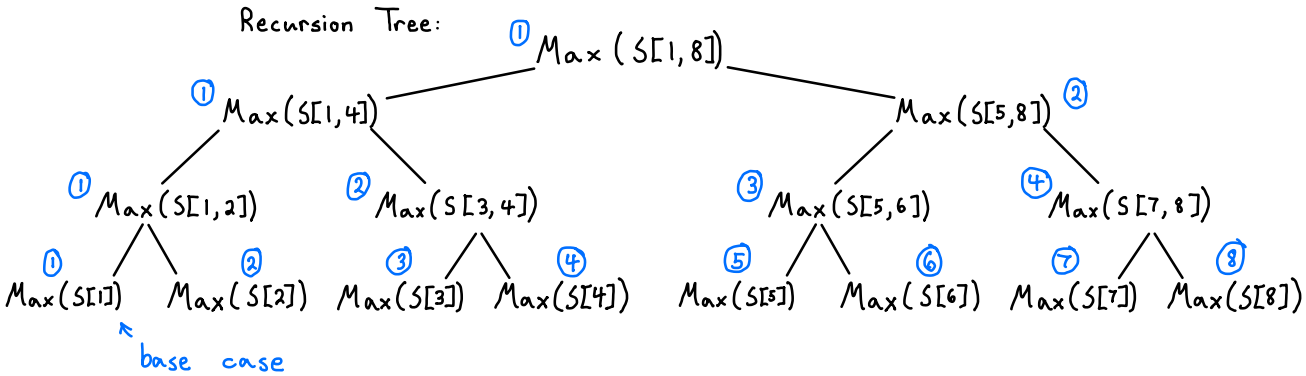
Time complexity : O(n)
- good time

Divide and Conquer Strategy

1. Split the problem into smaller independent subproblems
2. Recursively solve the subproblems
3. Combine the solutions of the subproblems to get the solution of the original problem

1. Split : array S ⟶ two (disjoint) equal parts

Recursion Tree:



2. Solve: find max for both parts (max_1 and max_2) recursively

3. Combine : maximum for S will be maximum between max_1 and max_2

Pseudo Code (start index, end index)

```
MAXR ( S, a, b):

    if  a = b                          } |S| = 1
        return S[a]                    }

midpoint ──→  m = ⌊(a+b) / 2⌋
1st half ──→  max1 = MAXR (S, a, m)        } ⌊T(n/2)⌋
2nd half ──→  max2 = MAXR (S, m+1, b)      } ⌈T(n/2)⌉

    if max1 > max2                     } 1
        return max1
    else
        return max2
```

- Call MAXR (S, 1, n) to solve the original problem

Proof by Induction

Base Case: $|S| = 1$

Algorithm works because there's only one element, so that's the maximum.

Inductive Step:

I.H. ⟶ Assume that the algorithm is correct for all sets of size less than n (1 ... n).

We want to show that the algorithm is correct for an input of n+1 elements.
(MAXR(S) is the max of S when $|S| = n$).

By the induction hypothesis, max_1 and max_2 are the correct max values of the $S[a \dots m]$ and $S[m+1 \dots b]$; since they are of size less than n.

Hence the maximum of max_1 and max_2 must be the maximum of $S[1 \dots n]$. □

Running Time

- Cost = number of comparison operations

  1 comparison operation : 1 recursive call

- Run time = number of recursive calls

- Recursive calls = size of the recursive tree or number of nodes

ex: $S[1 \ldots 8]$

$$|S[1 \ldots 8]| = 15 = 1 \cdot 2 \cdot 4 \cdot 8 = 2^0 \cdot 2^1 \cdot 2^2 \cdot 2^3 = 2^4 - 1$$

General format:

$$|S[1 \ldots 2^k]| = 2^0 + 2^1 + 2^2 + \ldots + 2^k = 2^{k+1} - 1 \leq 2(2^k)$$

$2n = O(n)$, where $n = 2^k$ is the size of the array

$O(n)$ = run time is proportional to the size of the array

Analyzing Recursive Algorithms

- Count the number of comparisons

$T(n)$ = number of comparisons on an input array size n for MAXR

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

With base case: $T(1) = 0$

- Solving Recurrence (Unwinding)

even size of n → Assume n is $n = 2^k$, for some k.

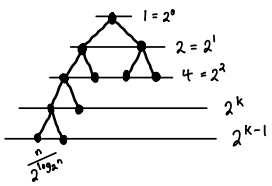
$$T(2^k) = 2T\left(\frac{2^k}{2}\right) + 1 = 2T\left(2^k \cdot 2^{-1}\right) + 1 = 2T\left(2^{k-1}\right) + 1$$

$$= 2\left(2T\left(2^{k-2}\right) + 1\right) + 1 = 2^2 T(2^{k-2}) + 2 + 1$$

$$= 2^2\left(2T\left(2^{k-3}\right) + 1\right) + 2 + 1 = 2^3 T(2^{k-3}) + 2^2 + 2 + 1$$

$$= 2^k T(1) + 2^{k-1} + \ldots + 1$$

$$= 2^{k-1} + 2^{k-2} + \ldots + 2 + 1 = \sum_{i=0}^{k-1} 2^i \quad \text{, where } k = \log_2 n$$



Geometric Series Reminder:

$$\sum_{i=0}^{k-1} x^i = 1 + x + \ldots + x^{k-1} = \frac{x^k - 1}{x - 1}$$ ← memorize

Therefore,

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

$$T(2^k) = 2^k - 1 = n - 1 = O(n)$$

Solving Recurrences

Guess and Verify

- Use induction to guess the correct solution
- Use induction (again) to prove that it is correct

Solve MAXR:

"Real" Recurrance is $T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 1$

Guess $T(n) = O(n)$.

① Show that $T(n) \leq cn$, for some constant
I.H.: Assume that $T(k) \leq ck$ for all $k \leq n$.

$$T(n) \leq c\lfloor \tfrac{n}{2} \rfloor + c\lceil \tfrac{n}{2} \rceil + 1$$
$$= cn + 1$$

which doesn't prove our hypothesis.

② Show that $T(n) \leq cn - b$, $\forall\, n \geq 1$, for some constant c and b.
I.H.: Assume that $T(k) \leq ck - b$ for all $k < n$.

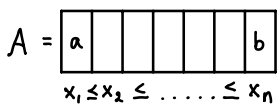
$$T(n) \leq \left(c\lfloor \tfrac{n}{2} \rfloor - b\right) + \left(c\lceil \tfrac{n}{2} \rceil - b\right) + 1$$
$$= cn - 2b - 1$$
$$cn - 2b - 1 \leq cn - b, \text{ if } b \geq 1$$

In MAXR, $T(1) = 0$. We choose $c = 1$ and $b = 1$, which satisfies the base case.
Thus, we have shown $T(n) \leq n - 1$ for all n.

Module 2: Divide and Conquer

Chapter 5.1: Sorting

Sorting Problem



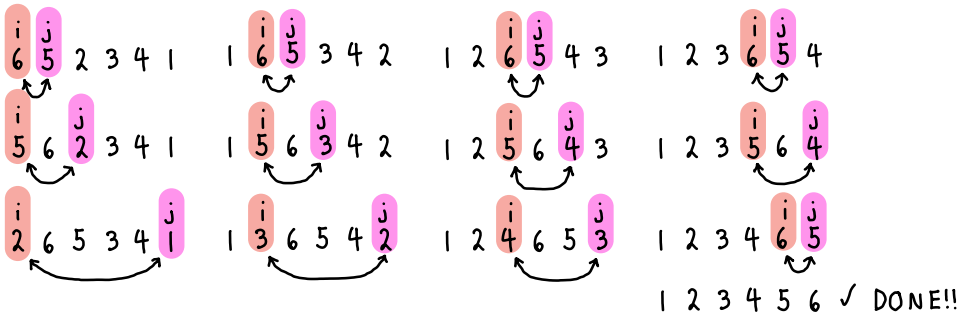
"Is $x_1 \leq x_2$?"

We want make an algorithm with the least comparison operations as possible.

Goal: $O(n \log(n))$ comparisons

Simple Sort

- Find the first and second smallest element, then swap positions



PseudoCode
function SIMPLESORT(A):

```
for i = 1 to n
    for j = i+1 to n
        if A[j] < A[i]
            swap(A[i], A[j])
```

$\longrightarrow$ $n \cdot n$ comparisons $= \frac{n(n+1)}{2} = O(n^2)$

Time

$n = 2^{30}$

n^2 or $\log(n)$

$(2^{30})^2$ $\quad$ $\log(2^{30})$

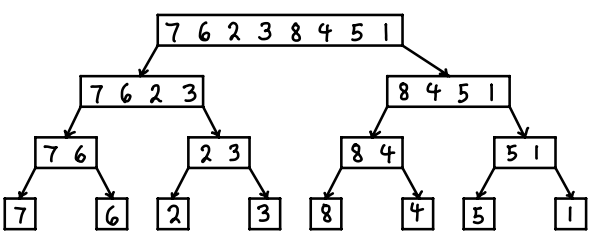
$2^{60} > 30\log(2)$, so $O(n \log n)$ comparisons is more efficient!!

Merge Sort

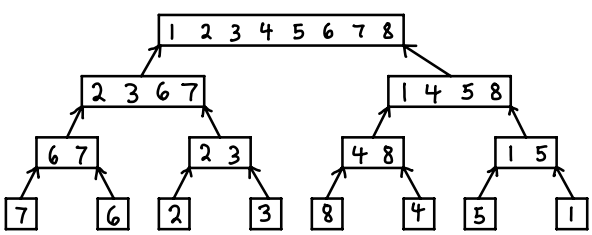
- Divide and Conquer Strategy

1. Split the array into two equal halves
2. Recursively sort the two subarrays independently
3. Combine both arrays by merging into one sorted array

Top-down Recursion

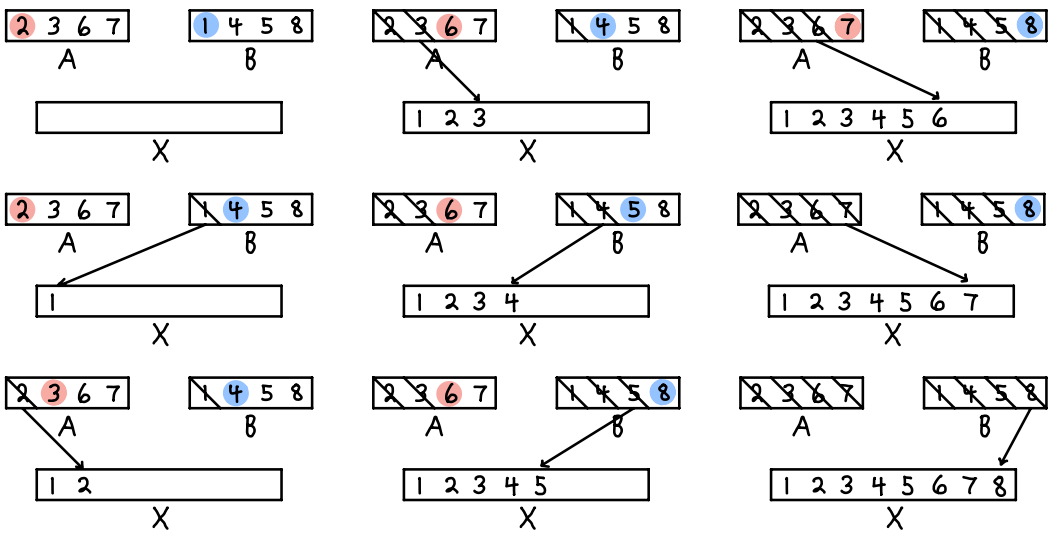


Bottom-up Merging



Merge

- Takes as input two sorted arrays A and B
- Outputs a merged sorted array X
- X is initially an empty array



Run Time

Merge: If A and B have sizes of k_1 and k_2, then there will be no more than $k_1 + k_2$ comparisons

↳ one comparison per adding one new element to X

Merge Sort: Assume n is a power of 2

number of comparisons $\rightarrow T(n) \leq \underset{a}{2}T\left(\frac{n}{\underset{b}{2}}\right) + \underset{f(n)}{n}$, with base case: $T(1) = 0$

Using DC Recurrence Theorem, $T(n) = O(n \log n)$.

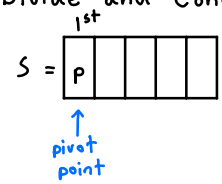
Pseudo Code

```
function MERGESORT(X, a, b):        # a = left, b = right

    if a < b
        mid = ⌈(a+b)/2⌉
        MERGESORT(X, a, mid-1)   # 1st half
        MERGESORT(X, mid, b)     # 2nd half
        MERGE(X, a, b)           # merge into X
```

```
function MERGE(X, a, b):

    i = a   # i = index of the 1st half

    mid = ⌈(a+b)/2⌉

    j = mid  # j = index of the 2nd half

    while i < j and j ≤ b
        if X[i] ≤ X[j]
            Temp[i+j] = X[i]   # copy from 1st half to Temp
            i = i+1
        else
            Temp[i+j] = X[j]   # copy from 2nd half to Temp
            j = j+1

    if j > b    # if second half finishes first
        for t = 0 to mid-1   # shift elements from 1st half to 2nd half
            X[b-t] = X[mid-1-t]

    for t = 0 to i+j-1
        X[a+t] = Temp[t]
```

Quick Sort

- Divide and Conquer Strategy



in-place sort p

1. Split the array into 2 sub arrays - partitioned based on p — $S_1 \leq [p] < S_2$
2. Recursively sort S_1 and S_2 (n-1 comparisons)
3. Concatenate all of the arrays — $S_1 + [p] + S_2$

Pseudo Code

```
function QUICKSORT(S):

    if |S| ≤ 1
        return S
    else
        p = S[1]   # 1st element of S is the pivot point
        S1 = { x ∈ S - {p} | x < p }
        S2 = { x ∈ S - {p} | x > p }
        return QUICKSORT(S1) + [p] + QUICKSORT(S2)
```

Run Time Analysis

- Worst-case is $T(n) = O(n^2)$

Proof. $T(n) \leq \max_{0 \leq k \leq n-1} \{ T(k) + T(n-k-1) \} + n - 1$

The pivot, k, partitions array of size, n, into 2 parts:
- S_1 of size k
- S_2 of size $n-k-1$

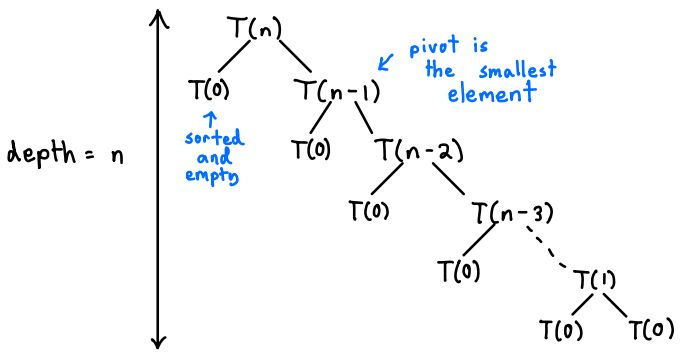
If the pivot is the median, then $k \approx \frac{n}{2}$

If the pivot is the largest element, then $k = n-1$ → $n-1$ comparisons

If the pivot is the smallest element, then $k = 0$ → $n-1$ comparisons

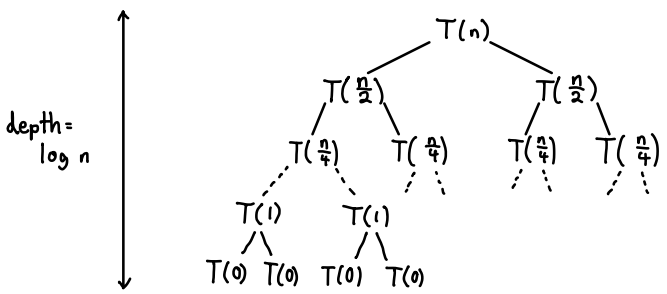
The worst case is when partition produces one subproblem of size $n-1$ and size 0, which leads to maximum imbalance

Since partition costs $n-1$ comparisons, $T(n) = T(0) + T(n-1) + n - 1$
$= T(n-1) + n - 1$
with $T(0) = 0$. Solving gives us $T(n) = O(n^2)$



The worst case is skewed to the right.

- Best-case is $T(n) = O(n \log n)$



The pivot is the median and there are equal two subproblems

By DC Recurrence Theorem, the recurrence is $T(n) \leq 2T(\frac{n}{2}) + n$, and $T(n) = O(n \log n)$

- Average-case is $O(n \log n)$

Chapter 5.2 Selection

Given an array S, we want to find the kth smallest element.

- Find a pivot point which will partion S into groups of 5 elements
- There will be $n/5$ medians for each group
- Recursively find the median of the $n/5$ medians
- "median of medians" is a good pivot point, that partitions S into appoximately balanced sets
- $|S_1| = k-1$, then pivot is the kth element

PseudoCode

```
function Selet(S, k):

    if |S| = 1
        return S[1]

    # Partition S into ⌊n/5⌋ groups of 5 elements each and a leftover
      group of up to 4 elements

    # Find the median of each group

    R = set of these ⌈n/5⌉ medians
    p = Select(R, ⌈|R|/2⌉)        ---> 7n/10 + 6 elements
    S1 = {x ∈ S | x < p}
    S2 = {x ∈ S | x > p}
    if |S1| = k-1
        return p
    else if |S1| > k-1
        return Select(S1, k)
    else
        return Select(S2, k - |S1| - 1)
```

Run Time

$p \geq S$

at least 3 elements $> p$ $\quad 3\left(\frac{1}{2}\lceil\frac{n}{5}\rceil - 2\right) \geq \frac{3n}{10} - 6$

($\frac{1}{2}$: half of the group)

The number of comparisons is bounded by αn, for some constant $\alpha n > 0$

$$T(n) \leq T\left(\left\lceil\frac{n}{5}\right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + \alpha n$$

By induction, assume $T(n) \le cn$ for some constant $c > 0$.

$$T(n) \le c\left(\frac{n}{5} + 1\right) + c\left(\frac{7n}{10} + 6\right) + \alpha n$$

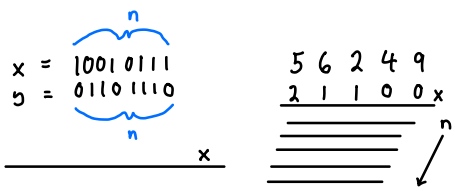
$$T(n) \le \frac{9cn}{10} + 7c + \alpha n$$

$$T(n) \le cn \quad , \text{ if } \quad n > 70$$

Thus, $T(k) = O(1)$, for $k \le 70$ ← base case

Chapter 5.3 Multiply two numbers

x and y be 2 n-bit numbers



$O(n^2)$, multiplication

Assume $n = 2^n$, where both numbers are split into $n/2$ bits

$$x = 2^{n/2} x_1 + x_2$$

$$y = 2^{n/2} y_1 + y_2$$

$$xy = 2^n (x_1 y_1) + 2^{n/2} (x_1 y_2 + x_2 y_1) + x_2 y_2$$

The recurrence is $T(n) = 4T(n/2) + O(n)$ with $T(1) = O(1)$.
By DC Recurrence Theorem, $T(n) = O(n^2)$.

Use Gauss' idea to improve Big-O,

$$(x_1 + x_2)(y_1 + y_2) = x_1 y_1 + x_1 y_2 + x_2 y_1 + x_2 y_2$$

$$x_1 y_2 + x_2 y_1 = \underbrace{(x_1 + x_2)(y_1 + y_2)}_{1} - \underbrace{x_1 y_1}_{2} - \underbrace{x_2 y_2}_{3}$$

The recurrence will be $T(n) = 3T(n/2) + O(n)$.
By the DC Recurrence Theorem, $O(n^{\log_2 3}) = O(n^{1.585}) < O(n^2)$ ✓

Chapter 5.5 Closest Pair

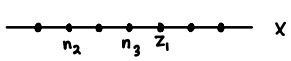
$\binom{n}{2}$ pairs $= \frac{n(n-1)}{2} = O(n^2)$

n points: $x = (x_1, x_2)$
$y = (y_1, y_2)$

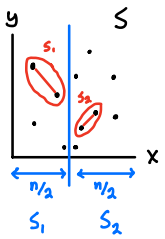
$$dist(x,y) = \sqrt{(x_1, x_2)^2 + (y_1, y_2)^2}$$

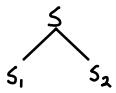
Solve with $O(n \log n)$:

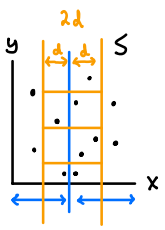
1D

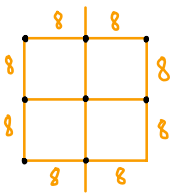


2D









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

$O(n \log n)$

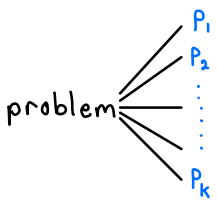
$T(n) = 2T(n/2) + O(n \log n)$

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

Main equation: $O(n \log^{d-1} n)$

Module 3: Dynamic Programming

General Idea



independent

not disjoint

overlapping

subproblems

Break the problem into multiple subproblems

Dynamic Programming = DP

ex: Fibonacci Numbers

0, 1, 1, 2, 3, 5, 8

F_n = nth Fibonacci

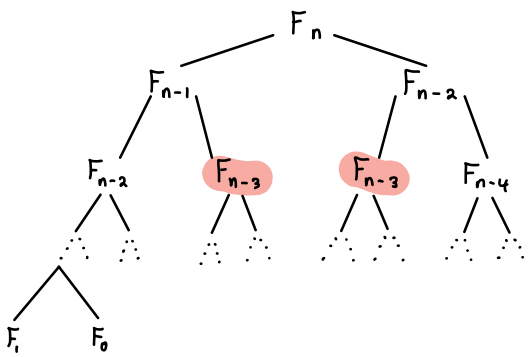
$F_0 = 0$

$F_1 = 1$

recursive $\longrightarrow F_n = F_{n-1} + F_{n-2}$ $\quad n \geq 2$

```
function Fib(n):
    if n ≤ 1
        return n
    else
        return Fib(n-1) + Fib(n-2)
```

Fib(100) is too big to terminate ☹



$$F_n \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n$$

↑ exponential time, not good x

F_{n-3} is overlapping, so solve it only once lol

use ↘

Table Look-Up Strategy

$F[0 \dots n] = \{-1\}$

```
function Fib(n):

    if F[n] ≠ -1
        return F[n]

    else if n ≤ 1
        return n

    else
        F[n] = Fib(n-1) + F(n-2)
        return F[n]
```

$O(n) < O(n!)$ SO MUCH FASTER!!!

Top down recursion

Bottom-Up

```
function Fib(n):

    F[0 ... n] = { 0, 1, 0, ... , 0}

    For i = 2 to n

        F[i] = F[i-1] + F[i-2]

    return F[n]
```

bottom up DP or iterative

Also, $O(n)$

Chapter 6.3 Maxsum Problem

$A = \{ 5, -1, 6, -2, -8, 8, 10, -9, 20 \}$ $n = 9$

$1 \le i \le j \le n$

Maximize : $A[i] + A[i+1] + \dots + A[j]$

Max value = 29

Feasible Solution

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

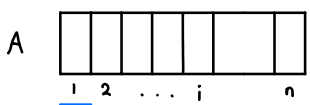
$A = \{ 5, -1, 6, -2, -8 \mid 8, 10, -9, 20 \}$

split in half → $O(n \log n)$

① Get the subproblems

- the smaller number of subproblems, the better

$M(n) \rightarrow$ maxsums for $A[1 \dots n]$



$M(i)$: maxsums for $A[1 \dots i]$ $1 \le i \le n$

$M(1) = A[1]$

$M^{in}(i) =$ max value for $A[1 \dots i]$ which includes $A[i]$

$M^{out}(i) =$ max value for $A[1 \dots i]$ which excludes $A[i]$

② Recursive

$M[i] = \max(M^{in}[i], M^{out}[i])$

$M^{in}[i] = \max(M^{in}[i-1] + A[i], A[i])$

$M^{out}[i] = M[i-1]$

$M^{in}[i] = A[i]$

$M^{out}[i] = -\infty$

③ Pseudo Code

```
function MAXSUMDP(A):

    M[1] = A[1]
    M^in[1] = A[1]

    for i = 2 to n

        M^ont[i] = M[i-1]
        M^in[i] = max(M^in[i-1] + A[i], A[i])
        M[i] = max(M^in[i], M^ont[i])

    return M[n]
```

Cache Table Strategy

- Another name for table look-up strategy

[-1, -1, 3, -2, 5, -1, 100, -1000]

min: -1 -1 3 1 6 5 105 -895
- add the previous element to the current element
- compare sums and take the minimum

table: F F F T T T T T
- find the largest sum and stop until the smallest positive sum is found

M^{in}: $A(i) = \max(A_{in}(i-1) + A[i], A[i])$
$A_{in}(0) = A[0]$

- Pseudo Code

```
cache = { 0 : A[0] }

function M_in(i):

    if i is not in cache:

        cache[i] = max(M_in(i-1) + A[i], A[i])

    return cache[i]
```

Chapter 6.4 Matrix Chain Multiplication

Example: $|A_1| = 10 \times 1000$
$|A_2| = 1000 \times 100$
$|A_3| = 100 \times 200$

$(A_1 A_2) A_3 = 10 \times 1000 \times 100 + 10 \times 100 \times 200 = 1,200,000$ costs

$A_1 (A_2 A_3) = 1000 \times 100 \times 200 + 10 \times 1000 \times 200 = 22,000,000$ costs

The recurrence is $P(n) = \sum_{k=1}^{n-1} P(k) P(n-k)$, $n \geq 2$

↑ number of different ways of parenthesizing a product of n matrices

$P(1) = 1$

$P(n) = \Omega(2^n)$, where $P(n)$ is exponential in n

For $1 \leq i \leq j \leq n$, $m[i,j]$ be the minimum cost of parathesizing the product $A_i A_{i+1} \dots A_j$

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{ m[i,j] + m[k+1, j] + p_{i-1} p_k p_j & i < j \end{cases}$$

Recursive formulation

Base Case: $i = j$ (aka. trivially true)

Recursive Step: $i < j$

The inductive hypothesis is that the recursive formulation is correct for all ℓ such $j - i < \ell$.

↑ length of the chain

Assume the hypothesis, prove that $j - i = \ell$
(aka $m[i,j]$ is the optimal cost for $A_i \dots A_j$).

$$(A_i \dots A_j) = (A_i \dots A_k)(A_{k+1} \dots A_j)$$

- $m[i,k]$ is the minimum cost of multiplying $(A_i \dots A_k)$
- $m[k+1, j]$ is the minimum cost of multiplying $A(A_{k+1} \dots A_j)$
- $p_{i-1} p_k p_j$ is the cost of combining the 2 partial products

Hence, since the split is minimum, the overall cost $m[i,j]$ is optimal.

The optimal substructure property tells us that there is no need to consider suboptimal solutions to subproblems when solving a problem.

- Only optimal solutions to subproblems are relevant.
- Drastically reduces the number of subproblem solutions to consider

↓

efficient algorithms

Bottom-up Algorithm

- DP / iterative algorithm
- first computes 1 length chains, then increase length (2, 3, ..., to n)
- each chain length l depends on $l-1$ or smaller

```
function MATRIX DP (p):   // sequence of matrix dimensions p

    for i = 1 to n:
        m[i,i] = 0

    for l = 2 to n:
        for i = 1 to n - l + 1:
            j = i + l - 1
            m[i,j] = ∞

            for k = i to j-1:
                q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
                if q < m[i,j]:
                    m[i, j] = q

    return m[1, n]
```

Top-down Algorithm

- recursive/table look-up
- avoid solving same subproblems take polynomial time

```
function MATRIX REC(p):
    return MATRIX HELPER(p, 1, n)

function MATRIXHELPEP(p, i, j):

    if m[i,j] = ∞:
        if i = j:
            m[i,j] = 0

        else:
            for k = i to j-1:
```

```
        left  = MATRIXHELPER( p, i, k)
        right = MATRIXHELPER( p, k+1, j)
        total = left + right + p_{i-1} p_k p_j

        if total < m[i,j]:
            m[i,j] = total

    return m[i, j]
```

Runtime

- Distinct subproblems for every pair of indices

  $1 \le i \le j \le n \longrightarrow O(n^2)$ subproblems

- Each subproblem can be solved in $O(n)$

Total time is $O(n^3)$

> Note:
> DP runtime is
> O(# subproblems x time per problem)

Chapter 6.5 Aligning 2 Sequences

Given 2 protein/DNA sequences,

s = GACGGATTAG m = |s|
t = GATCGGAATAG n = |t|

Find the best alignment between the "strings"

↳ sum of score, so best alignment = max score
called similarity ~ sim(s, t)

example:

$$score(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \\ -1 & \text{if } x \text{ or } y \text{ is a space} \end{cases}$$

```
s = GA CGGATTAG
t = GATCGGAATAG
    1 1-1 1 1 1 1 0 1 1 1
```

sim(s, t) = 1 + 1 - 1 + 1 + 1 + 1 + 1 + 0 + 1 + 1 + 1 = 8

Recursive Formulation

s = list of characters s[1 ... m]
t = list of characters t[1 ... n]

Subproblems: find the best alignment for substrings

s[1 ... i], where $1 \le i \le m$

t[1 ... j], where $1 \le j \le n$

sim(i, j)
↓ answer
sim(n, m), where
len(s) = n and
len(t) = m

Compute $sim(s[1\dots i], t[1\dots j])$ with only 3 possibilities:

1) align $s[i]$ with $t[j]$
2) align gap with $t[j]$
3) align $s[i]$ with gap

Recursive formula:

$$sim(s[1\dots i], t[1\dots j]) = \max \begin{cases} sim(s[1\dots i-1], t[1\dots j-1]) + score(s[i], t[j]) \\ sim(s[1\dots i], t[1\dots j-1]) + score(-, t[j]) \\ sim(s[1\dots i-1], t[1\dots j]) + score(s[i], -) \end{cases}$$

Base case:

$sim(s[1\dots i], t[\,]) = -i$ (1) when t is empty, align s with gaps

$sim(s[\,], t[1\dots j]) = -j$ (2) when s is empty, align t with gaps

$sim(s[\,], t[\,]) = 0$ (3) both are empty, same size already

Bottom-up (DP) Algorithm

```
function SIM DP(s, t):

    for i = 0 to m:
        a[i, 0] = -i

    for j = 0 to n:
        a[0, j] = -j

    for i = 1 to m:
        for j = 1 to n:
            both = a[i-1, j-1] + score(s[i], t[j])
            left = a[i, j-1] + score(-, t[j])
            right = a[i-1, j] + score(s[i], -)
            a[i, j] = max(both, left, right)

    return a[m, n]
```

Time Complexity: $O(mn)$

Chapter 6.6 0/1 Knapsack

Given $A = \{a_1, a_2, \dots, a_n\}$ with n objects and a knapsack

- a_i has a weight w_i and knapsack with capacity m
- add a_i into knapsack if $a_i \le m$
  - profit p_i is earned
- omit a_i if $a_i > m$

x_i { 1 if added / 0 if omitted

Goal: maximize p_i with knapsack constraint

Subproblems

- Use auxiliary variables (two new variables j, y)
- KNAP(j, y) represents $\sum_{1 \le i \le j} p_i x_i$ profits with constraint $\sum_{1 \le i \le j} w_i x_i \le y$, where $x_i \in \{0, 1\}$, $1 \le i \le j$
  - Basically consider only the first j items with knapsack capacity y

Recursive Formulation

$P_j(y)$ = optimal solution to KNAP(j, y)

$$P_j(y) = \max(P_{j-1}(y), P_{j-1}(y - w_j) + p_j)$$

Base Case:

$P_0(y) = 0$ for all $y \ge 0$

$P_i(y) = -\infty$ if $y < 0$

Correctness

To compute $P_j(y)$, we have 2 choices:

1) a_jth is in the knapsack

$P_j(y) = P_{j-1}(y - w_j) + p_j$

- subtract weight w_j from capacity
- included profit

2) or not

$P_j(y) = P_{j-1}(y)$

- go to the previous item as is

Through induction, optimal substructure property is used.

$P_j(y)$ will contain optimal solution to the subproblems:

1) $P_{j-1}(y)$
2) $P_{j-1}(y - w_j)$

Take the max value from the solutions

DP Algorithm

- optimal value $P_n(m)$
  - KNAP(n, m) → bottom-up algorithm
  - start from base cases
  - then, compute $P_1(.)$, $P_2(.)$, ... in order
- runtime is $O(nm)$
  - depends on input size, not input
    - ↘ NOT polynomial

      m is represented using log m bits:
      $b = \log m \rightarrow O(nm) = O(n\,2^b)$
  - known as pseudo-polynomial
- known as a pseudo-polynomial algorithm

Module 4: Greedy Algorithms

Chapter 7.1 Fractional Knapsack

Given n objects with: weights = $w_1, w_2, \ldots, w_n$

profits = $p_1, p_2, \ldots, p_n$

Knapsack with capacity m, choose:

$x_1, x_2, \ldots, x_m$ (between 0 and 1)

Maximize: $\sum_{i=1}^{n} x_i p_i$ With constraint: $\sum_{i=1}^{n} x_i w_i \leq m$

Outline:

- allowed to choose a fraction of an object
- if $\sum_{i=1}^{n} w_i \leq m$, then $x_i = 1$

  $1 \leq i \leq n$ is an optimal solution
- All optimal solutions will the knapsack EXACTLY

Greedy Strategies

$n = 3$ and $m = 20$

Item	Weight	Profit
I_1	18	25
I_2	15	24
I_3	10	15

Solutions / "greedy" strategies to the problem:

1) fill knapsack by including the next object with largest profit
   - if object doesn't fit, then include a fraction of it to fill the knapsack
2) fill knapsack by including objects in non-decreasing order of weights
3) have objects in non-increasing order of $\frac{p_i}{w_i}$

Different Solutions Table

Quantity			Total	
I_1	I_2	I_3	Weight	Profit
1/2	1/3	1/4	16.5	24.25
1	2/15	0	20	28.20
0	2/3	1	20	31.00
0	1	1/2	20	31.50

- Using strategy 3, the max profit is 31.50
  - implemented in $O(n \log n)$ time (sort in non-increasing order)

Only consider one subproblems at a time

- for strategy 3, we choose the object with the hightest profit to weight ratio among the items that are NOT YET chosen

Proof of Optimality

Theorem: If objects are included in the non-increasing order of p_i/w_i, then this gives an optimal solution to the knapsack problem

General Proof Technique:

4) $Max\text{-}PROD(i)$ = max product in $A[1 \dots i]$

$Max\text{-}PROD_{in}(i)$ = max product in $A[1 \dots i]$, $1 \le i \le n$

$Min\text{-}PROD_{in}(i)$ = min product in $A[1 \dots i]$, $1 \le i \le n$

a) $Max\text{-}PROD(n)$

b) $Max\text{-}PROD_{in}(1) = A[1]$

$Min\text{-}PROD_{in}(1) = A[1]$

c)
$A[1 \dots i]$

$$Max\text{-}PROD_{in}(i) = \max\begin{Bmatrix} Max\text{-}PROD_{in}(i-1) \cdot A[i], \\ Min\text{-}PROD_{in}(i-1) \cdot A[i], \\ A[i] \end{Bmatrix}$$

$$Min\text{-}PROD_{in}(i) = \min\begin{Bmatrix} Max\text{-}PROD_{in}(i-1) \cdot A[i], \\ Min\text{-}PROD_{in}(i-1) \cdot A[i], \\ A[i] \end{Bmatrix}$$

d) $2n$ subproblems = $O(n)$

e) $O(1)$

f) $O(n)$

3) $[1, 1, 0, 1, 1]$

a)

b) return left and right and onederful(__) and onederful(__)

c) $T(n) = 2T(\frac{n}{2}) + O(1)$

d) $T(n) = O(n)$

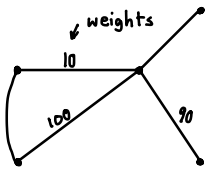
Hi Enlai
- Chris :P

LENA ♡
thank you for your help! :)

Module 5: Graphs / Networks

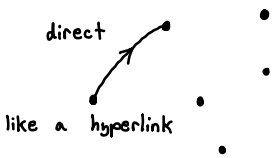
$G = (V, E)$

Graph = (Vertices, Edges)
↓ ↘
Nodes links



Directed / Undirected
Weighed / Unweighed

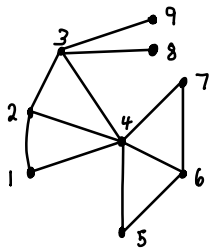
Web Graph



Used in a search engine:

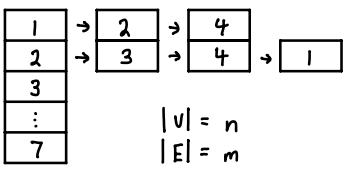
- Connects web links to web links

How to store a graph



$\deg(v)$ = # weights of V
$\sum$ degrees$(v) = 2m$
$w \in V$

- Use an adjacent list
  - space efficient



$|V| = n$
$|E| = m$
$O(n+m)$

$n + 2m$

- Use adjacent matrix

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	1
4	1	1	1	0

"sparse" graph

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

Depth-first Search (DFS)

- Implemented in $O(|V| + |E|)$
- Undirected graph

Steps:

- start at source vertex (s) and explore edges via depth-first
  - explore from most recently visited vertex (v)

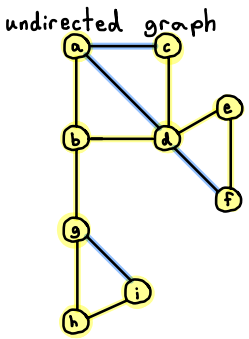
- If the explored edge leads to an already visited vertex, then another edge of v is explored
- If all neighbors of v have been visited, then the search backtracks to the vertex that was visited prior to v
- For iterative implementation, use a stack to backtrack

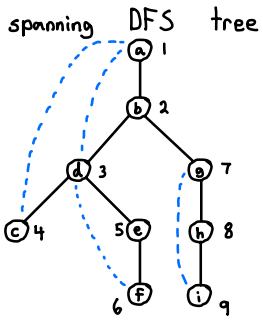
```
function DFS(G, s):

    visited = True
    num[s] = count
    count += 1
    for vertex v ∈ adj[s]:
        if not visited[v]:
            add edge(s, v) to T
            DFS(G, v)
```

visited initialized to false for every $v \in V$
- checks whether the vertex has been visited or not

num[v] gives the DFS number of v
- the order v was visited in the graph

tree T which stores edges
- has exactly n-1 edges at the end





tree edge

back edge

Runtime

- The number of recursive calls = if statement
  - each vertex is visited exactly once because the visited array will mark a vertex true when visited → $O(n)$
- The if statement will execute as the for loop runs
  - runs degree of v times ⇒ number of neighbors of v
  - total iteration: $\sum_{v \in V} \deg v = 2m$

Total runtime is $O(n + m)$

Key properties

1) tree edges: edges that belong to T
2) back edges: edges that don't belong to T
   - an edge that connects a vertex to a vertex that is discovered before it's parent

Property of back edges:
If (v, w) is a back edge, then in a DFS spanning tree T, v is an ancestor of w or vice versa.

Connected components of a graph

Given a (possibly) disconnected $G = (V, E)$, check if the graph is connected.

Goal: Find the connected component of a graph in linear time.

connected component: a subgraph in which each pair of nodes is connected with each other via path

```
function DFSConnectedComponents(G):
    for v ∈ V:
        visited[v] = False

    for v ∈ V:
        if not visited[v]:
            DFS(G, v)
```

Each DFS(G, v) call will return a connected component of G

If connected, then only one component is returned

DFS will explore and mark all nodes as visited in the first DFS call

If disconnected, each DFS call will return the component containing the vertex where the DFS is called.

Cut vertices and cut edges of a graph

Let $G = (V, E)$ be a connected undirected graph

Vertex x is said to be a cut-vertex of G:

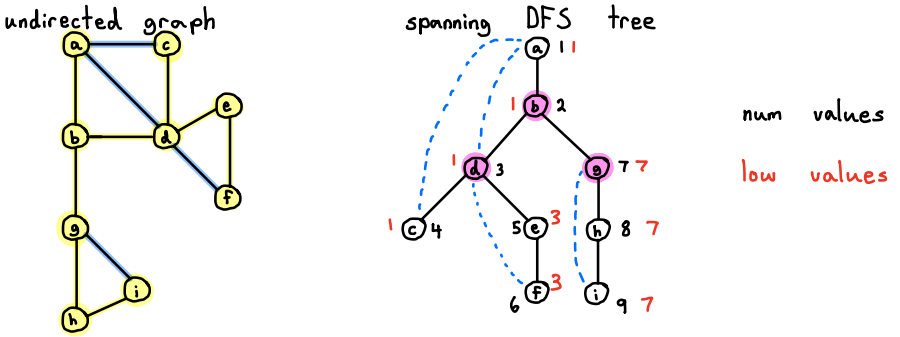
- If removing x from G disconnects the graph
- If there exists two other vertices u and v contains the vertex x (x, u, and v can't be in a cycle)

A graph is bi-connected:

- If the graph has no cut vertices

Edge $e \in E$ is a cut-edge of G:

- If removing e disconnects the graph
- A cut-edge cannot lie in any cycle of the graph



The cut vertices of G are : b, d, g

Note :

- Each of their num values is less than or equal to the low values of at least one their respective children
  - d is a cut vertex because e's low value greater than or equal to d's num value
- Cut vertex will not have a back edge that goes from its descendent to its ancestor
  - d is a cut vertex, since it's the only vertex that connects its descendent f to its ancestors

Using DFS to find all cut-vertices

- Use back edge property to characterize a cut vertex

> Let T be a DFS spanning tree of G. A vertex x is a cut vertex of G if and only if :
>
> 1) x is the root of T and x has more than one child
>
> 2) x is not the root, and for some child of y of x, there is no back edge between any descendant of y (including y itself) and a (proper) ancestor of x

- Condition 1 is simple
- Condition 2 involves DFS and computing low[]

Define low[v] as : $low[v] = \min\{num[v], back\text{-}edge\text{-}num[v]\}$

where $back\text{-}edge\text{-}num[v] = \min\{num[w] \mid \exists$ a back edge (x, w) such that x is a descendant of v (including v)$\}$

↓

highest ancestor that any descendant of v can reach

low values can be done recursively:

since low value of a vertex x depends on its num value, the low value of its children and whether x has any back edge

$$low[x] = \min\{\, num[x],\ \{low[y] \mid y \text{ is a child of } x\},\ \{num[w] \mid (x,w) \text{ is a back edge}\}\,\}$$

Condition 2 is satisfied for x if it has a child y such that $low[y] \geq num[x]$

```
function DFSCutVertices(G, s):

    visited[s] = True
    num[s] = count
    count += 1
    low[s] = num[s]

    for v ∈ adj[s]:
        if not visited[v]:
            add edge (s,v) to T
            parent[v] = s
            DFSCutVertices(v)
            if low[v] ≥ num[s]:
                output s is a cut-vertex
            low[s] = min(low[s], low[v])
        # checks if (s,v) is back-edge
        else if v ≠ parent[s]:
            low[s] = min(low[s], num[v])
```

- Directed graph

Classification of edges in a directed graph

1) tree edges: edges that belong to the DFS tree
   - if v is visited for the first time when exploring edge (u, v)

2) back edges: edges that connecting a vertex to its ancestor

3) forward edges: edges that connect a vertex to its descendent

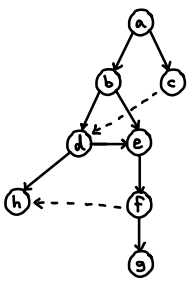
4) cross edges: all other edges of G.
   - they can go between vertices in the same DFS tree or in different trees

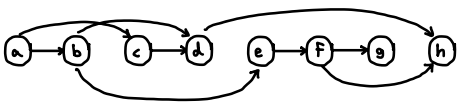
A directed graph is strongly connected if for every pair of nodes u and v, there is a directed path from u to v and from v to u.

Called weakly connected if the graph is connected by ignoring directions.

Directed Acyclic Graph (DAG)

- DAG is a directed graph with no (directed) cycle.
- Direct edges: (u, v) denotes that operation u has to be performed before operation v
- Topological sort
  - gives a valid ordering of operations (precedence constraints are obeyed)
  - all directed edges go from left to right
  - if a DAG, then there is topological sort





- There are no back edges

```
function DFSFinishTimes(G):

    for v ∈ V:
        visited[v] = False
    time = 0
    for v ∈ V:
        if not visited[v]:
            DFSFinishTimeHelper(v)

function DFSFinishTimeHelper(s):

    visited[s] = True
    time += 1
    num[s] = time
    for v ∈ adj[s]:
        if not visited[v]:
            DFSFinishTimeHelper(v)
    time += 1
    fin[s] = time
```

The function computes finishing time for vertices

Recall: num[v] = order in which vertices were visited

Let fin[v] denote the order in which v was finished processing
↘
the time at which the recursive call DFS(v) returns

```
function TopoSort(G):
    DFSFinishTimes(G)
    Output vertices by decreasing order of their finish times
```

use an array and return vertices in reverse order

Runtime: $O(|V| + |E|)$ → same as DFS

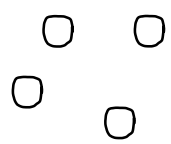
- Key properties of a DAG
  - In a DAG G, DFS yields no back edges
  - TopoSort algorithm produces a topological ordering of a DAG

Breadth-first search (BFS)

- Start from vertex s and visit vertices in increasing order of distance from s
- Find the shortest path / distance from s to all other vertices

distance between two vertices = # of edges

- Implemented in $O(|V| + |E|)$



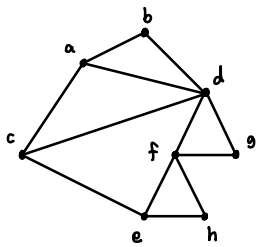
function DJ

```
For v ∈ G:
    d[v] = ∞
    π[v] = null

d[s] = 0
S = ∅
Q = Queue(G)
while Q is not empty:
    u = Q.ExtractMin()
    S.Add(u)
    for v ∈ adj[u]:
        newdist = d[u] + w(u, v)
        if newDist < d[v]:
            Q.decreaseKey(v, newDist)
            π[v] = u
```

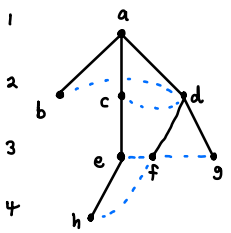
```
for v ∈ G:
    d[v] = ∞
    π[v] = null

d[s] = 0
S = ∅
Q = queue(G)
while Q not empty:
    u = Q.extractmin()
    S.add(u)
    for u ∈ adj[u]:
        newdist = d[u] + w(u, v)
        if newDist < d[i]
            Q.decreasekey(v, newdist)
            π[
```

BFS

- UNDIRECTED graph



BFS spanning tree



level by level

distance = length of the shortest path

diameter of the graph = max distance between 2 nodes

$$D = \max_{u,v} dist(u, v) \leq h - 1$$

Bellman - Ford

$$d_k(v) = \min_{\substack{u \text{ is a} \\ \text{neighbor} \\ \text{of } V}} \{ d_{k-1}(u) + w(u, v) \}$$

(u, v)

$d_0(v) = \infty$ if $v \neq s$

$d_0(s) = 0$

matrix

$$\begin{array}{c} \\ u=0 \\ u=1 \\ \vdots \\ u=n \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & \cdots & v_n \end{array} \\ \begin{bmatrix} \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ & & & \\ & & & \end{bmatrix} \end{array}$$

Dijkstra ($G = (V, E)$, S)

for all $v \in V$

$d[v] = \infty$
$\pi[v] = \perp$

run time:
$O(m + n \lg n)$

$d[s] = 0$
Q = build heap (v)
while Q is not empty

u = ext (Q)

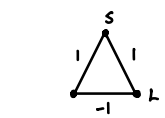
for add $v \in Adj(u)$

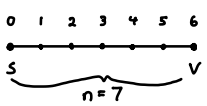
if $d[v] > d[u] + w(u, v)$

distra $(Q, v, d[u] + w(u,v))$
$P[v] = u$

- main weakness
  - doesn't work w/ negative weights
- good for numerical computation

Bellman - Ford (DP)





up to $n-1$ edges

SSSP

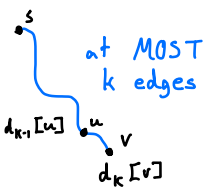
$d[v]$: distance for S to V
$d[1 \dots n] = ?$

1) subproblems

$d_k[v]$ = distance from s to v using at most k edges, where $0 \le k \le n-1$

$d[v] = d_{n-1}[v]$

2) recursion



base case: $d_0[v] = 0$ if $v = s$

$= \infty$ if $v \neq s$

recursion:
$$d_k[v] = \left(\min_{u \in Adj(v)} \left(d_{k-1}[u] + w(u, v) \right), d_{k-1}[v] \right)$$

3) code

```
BF( G=(V,E), s):
O(n)  for v ∈ V
          d[v] = ∞
      d[s] = 0

      for k=1 to n-1 } O(n)
O(m)      for all (u,v) ∈ E
m edges       if d[u] + w(u,v) < d[v]
                  d[v] = d[u] + w(u,v)

O(n)  for all (u,v) ∈ E
          if d[u] + w(u,v) < d[v]
              output " - v
```

Run time:
$O(nm)$