Week 1

Abdollahi Notes

→ Finding average case is not always possible to find.

→ Cases are not related to notations. Notations are for the functions.

→ Cases are just type of analysis done on algorithms.

CASES

Ex: $BestCase(n) = B(n) = 1$

$B(n) = O(1)$

$B(n) = \Omega(1)$

$B(n) = \Theta(1)$

Ex: $WorstCase(n) = n$

$W(n) = O(n)$

$W(n) = \Omega(n)$

$W(n) = \Theta(n)$

It belongs to constant class.

Best case can be written using any notation.

Worst case can be written using any notation.

Average case can be written using any notation.

Binary search tree ⇒ Height = $\log n$

Reflexive: R on A is reflexive when $(a,a) \in R$ for every element $a \in A$.

Symmetric: For all (x,y) values, there must be (y,x) values by satisfying $y \neq x$.

Anti-symmetric: If $\forall a \forall b \quad ((a,b) \in R \wedge (b,a) \in R) \Rightarrow (a=b)$

Transitive: Whenever $(a,b) \in R$ and $(b,c) \in R$, then $(a,c) \in R$ for all $a,b,c \in A$ and $a \neq b \neq c$.

Average Case Complexity

$$A(n) = \sum_{I \in I_n} T(I) \cdot P(I)$$

$$= T(I_1) \cdot P(I_1) + T(I_2) \cdot P(I_2) + \dots$$

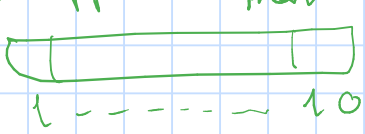
$$= \sum_{i=1}^{w(n) \to \infty} \frac{i \cdot P_i}{I}$$

I: # of operations

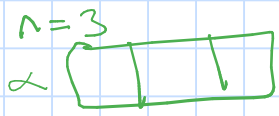
P_i: Prob that the algorithm performs i basic operations.

Ex: An algorithm for searching in a list of size n

→ Suppose that $n = 10$



$n = 3$

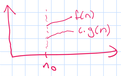


3! $B(n)$ is very good

$A(n)$ is very bad

o Notation

A function $f(n) \in w(g(n))$ iff for every positive constant c, there is a positive integer n_0 such that $c \cdot g(n) < f(n)$ whenever $n \geq n_0$.



$f(n)$ grows faster than $g(n)$

Ex: $n^3 \in w(n^2)$

$n^2 \notin w(5n^2)$

Analogy Between Asymptotic Notation - Relational Operator

$\Leftrightarrow$ = is like equality. Grow rate are same.

$O \leftrightarrow \leq$ lower bound, most frequently used

$\Omega \leftrightarrow \geq$

$o \leftrightarrow <$

$w \leftrightarrow >$

$f(n,m) \in O(g(n,m))$ iff there exists positive constants c and n_0 such that $f(n,m) \leq c \cdot g(n,m)$ whenever both $n, m \geq n_0$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0, & \text{if } f(n) \in o(g(n)) \\ c, \text{ where } c \geq 0 & \text{if } f(n) \in \theta(g(n)) \Rightarrow \text{constant time max } \theta \text{ class} \\ \infty, & \text{if } f(n) \in w(g(n)) \Rightarrow \frac{n^2}{n} \text{ comes to infinity, strict lower bound} \\ 1, & \text{if } f(n) \in \sim(g(n)) \end{cases}$$

not important

↳ tilda (always asymptotic)

- if $g(n)$ grows faster, it mean strict upper bound. little o notation.

Disadvantage of Limit Approach

- limit is not always exist.

L'Hopital's Rule

Let $f(x)$ and $g(x)$ be functions that are differentiable for sufficiently large real numbers x. If $\lim_{n\to\infty} f(x) = \infty$ and $\lim_{n\to\infty} g(x) = \infty$ OR if $\lim_{n\to\infty} f(x) = 0$ and $\lim_{n\to\infty} g(x) = 0$, then

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

Properties: Let $f(n), g(n) \in f_{\cup}$ set of functions

$\Rightarrow f(n) \in O(g(n)) \Leftrightarrow g(n) \in \sim(f(n))$ iff

If $g(n)$ is an upperbound to $f(n)$, then $f(n)$ is a lowerbound for $g(n)$

$\Rightarrow f(n) \in \theta(g(n)) \Rightarrow g(n) \in \theta(f(n))$ Same complexity class

$\Rightarrow f(n) \in \theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

↳ it's limitor to equal

$\Rightarrow f(n) \in o(g(n)) \Leftrightarrow g(n) \in w(f(n))$

$\Rightarrow f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$

Reverse is not necessarily true. For ex: $2 < 3 \Rightarrow 2 \leq 3$

$\Rightarrow f(n) \in \Omega(g(n)) \Rightarrow f(n) \in w(g(n))$

* Given $c > 0$; $\theta(f(n)) = \theta(c \cdot f(n))$

c is positive constant

c doesn't matter in complexity class,

- Also true for other complexity classes except $\sim$ not important

* $f(n) \in O(g(n)) \Leftrightarrow O(f(n)) \subseteq O(g(n))$

Ex: $n \in O(n^2)$ Because of low order terms in set.

$O(n) \subseteq O(n^2)$

* $f(n) \in o(g(n)) \Leftrightarrow O(f(n)) \subset O(g(n))$

↳ proper subset

* $O(f(n)) = O(g(n)) \Leftrightarrow \theta(f(n)) = \theta(g(n))$

$\Leftrightarrow \sim(f(n)) = \sim(g(n))$

---

* Given 2 functions $f(n), g(n) \in F$, $f(n)$ and $g(n)$ have the same order if $f(n) \in \theta(g(n))$

* $f(n)$ has the smaller order than $g(n)$ if $O(f(n)) \subset O(g(n))$ [$f(n) \in o(g(n))$]

* $f(n)$ and $g(n)$ are not comparable if $f(n) \notin O(g(n))$ and $g(n) \notin O(f(n))$.

Ex: Functions involving modulo operators. $\rightarrow$ periodic or cyclic behaviour

$\theta(n) = \ldots\ldots (\text{mod } 2)$

Popularly Used Classes

Fastest $\rightarrow O(1)$ constant order

Very very fast $\rightarrow O(\log \log n)$, $O((\log \log n)^{1.9})$ poly-logarithmic order

$\rightarrow O(\log n)$ logarithmic order (when binary search)

* Base of the logarithm is irrevelant in algorithm analysis.

$\rightarrow O(n^a)$ for $0 < a < 1$ $\Rightarrow$ most famous of this class is $\theta(\sqrt{n})$

Ex: Prove which is faster for $0 < a < 1$; $O(n^a)$ or $O(\log n)$

n^a grows faster than $\log n$ $\quad 0 < a < 1$

Consider $a = 0.01$

$\Rightarrow \lim_{n\to\infty} \frac{n^{0.01}}{\log n} = \frac{\infty}{\infty}$ L'Hopital

$\Rightarrow \frac{0.01 \times n^{-0.99}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{0.01 \times n^{0.01}}{(a \times n^a)} = \infty$

$n^{0.01}$ grows faster than $\log n$

$\Rightarrow n^{0.01} \in w(\log n)$

Notation for functions

$\rightarrow O(n)$: Linear order

$\rightarrow O(n \log n) \Rightarrow$ Typical in divide-and-conquer algorithms.

e.g.: merge sort, quicksort

$\rightarrow O(n^2) \Rightarrow$ Quadratic order

$\rightarrow O(n^3) \Rightarrow$ Cubic order

$\rightarrow O(a^n)$ for $a > 1 \Rightarrow$ Exponential order $\frac{d}{dx} 2^x = 2^x \cdot \log 2$

* a^n grows faster than n^a (means program a^n will work slower).

- Complexity of Towers of Hanoi is 2^n

* $a^{\log n}, a^{\frac{1}{\log n}}$ are also exponential. The power need not be n.

if power is log, may grow and may shrink (Poly-logarithmic)

2^n is typical algorithms that generate all possible subsets of a given set. Exponential algorithms are very slow.

$\rightarrow O(n \cdot a^n)$; $a > 1$

$\rightarrow O(n!) \Rightarrow$ Factorial order (very very slow)

$\rightarrow O(a^{n^b})$, for $a, b > 1$

$O(1) \subset O(\log\log n) \subset O(\log n) \subset O(n^a) \subset O(n) \subset O(n\log n) \subset O(n^a) \subset O(a^n) \subset O(n \cdot a^n) \subset O(a^{n^b})$

for $0 < a < 1$ (under $O(n^a)$); $a > 1$ (under the second $O(n^a)$)

Polynomial ← | → Exponential

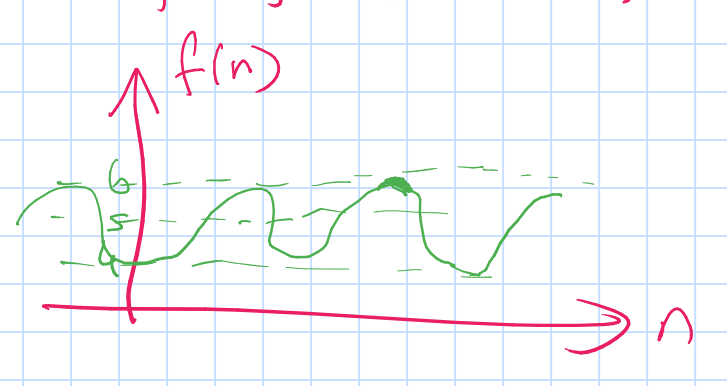
- An algorithm which it's complexity is at most $O(f(n))$ where $f(n)$ is a polynomial of n is called a polynomial time algorithm.

- An algorithm which it's complexity is at least $\Omega(f(n))$ where $a > 1$, is called an exponential time algorithm.

This distinction is the most important thing in the running time of algorithms.

⇒ Polynomial algorithms are feasible algorithms, whereas exponential algorithms are infeasible.

Ex: $f(n) = 5 + \sin(n)$



Both upperbound and lowerbound is constant.
Banded above by 6,
bounded below by 4.

$f(n) \leq 6$ so $f(n) \in O(1)$
$f(n) \leq c \cdot 1$ for $n \geq n_0$

$f(n) \geq 4$ so $f(n) \in \Omega(1)$
because $c \cdot 1 \leq f(n)$ for $n \geq n_0$ ($c = 4$)

So; $f(n) \in \Theta(1)$

Ex: Prove that $\sum_{k=1}^{n} k^p$ is $\Theta(n^{p+1})$ for $p \geq 1$ a fixed constant.

First, find an upper bound.

$$\sum_{k=1}^{n} k^p = \underbrace{1^p + 2^p + 3^p + \dots + n^p}_{n \text{ terms}} \leq n \cdot n^p = n^{p+1}$$

Each term is at most n^p ($\leq n^p$)

$\Rightarrow \sum_{k=1}^{n} k^p \in O(n^{p+1})$

Find a lower bound:

$$\sum_{k=1}^{n} k^p = \underbrace{\sum_{k=1}^{\lfloor n/2 \rfloor} k^p}_{A} + \underbrace{\sum_{k=\lfloor n/2 \rfloor+1}^{n} k^p}_{B} \geq \underbrace{\sum_{k=\lfloor n/2 \rfloor+1}^{n} k^p}_{B}$$

$\lfloor n \rfloor$ Floor function

$\lceil n \rceil$ Ceiling function

I want lower bound high as possible
(Take bigger term as lower bound)

⇓

$$\underbrace{(\lfloor \tfrac{n}{2} \rfloor + 1)^p + (\lfloor \tfrac{n}{2} \rfloor + 1)^p + \dots + n^p}_{\text{There are } n - \lfloor n/2 \rfloor = \lceil n/2 \rceil \text{ term}}$$

Each term $\geq (\lfloor \tfrac{n}{2} \rfloor + 1)^p \geq (\tfrac{n}{2})^p$

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

$\therefore \sum_{k=1}^{n} k^p \in \Omega(n^{p+1})$

Ex: $f(n) = \frac{n(n+1)}{2}$ find the asymptotic relationship btwn 2 functions, $g(n) = n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2+n}{2n^2} = \frac{1}{2} \Rightarrow f(n) \in \Theta(g(n))$$

Ex: Show that any polynomial function has smaller order than any exponential function i.e., $n^k \in o(a^n)$, or in other words, $O(n^k) \subset O(a^n)$ for $k > 0$, $n > 1$.

$$\lim_{n\to\infty} \frac{n^k}{a^n} = \lim_{n\to\infty} \frac{k \cdot n^{k-1}}{a^n \cdot \ln a} = \lim_{n\to\infty} \frac{k(k-1) \cdot n^{k-2}}{(\ln a)^2 \cdot a^n}$$

L'Hopital ($\ln a$: Constant)

Take derivative consequently. L'Hopital

$$= \lim_{n\to\infty} \frac{k(k-1) \dots 1}{(\ln a)^k \, a^n}$$

$$= \lim_{n\to\infty} \frac{k!}{(\ln a)^k \cdot a^n} = 0$$

(a^n: Always dominant here)

Ex: Compare orders of growth of $n!$ and 2^n.

$$\lim_{n\to\infty} \frac{n!}{2^n}$$

Stirling's formula → just plug in.
$n! \cong \sqrt{2\pi n} \cdot (\frac{n}{e})^n$ for large n.

$$= \lim_{n\to\infty} \frac{\sqrt{2\pi n} \cdot (\frac{n}{e})^n}{2^n} = \lim_{n\to\infty} \sqrt{2\pi n} \cdot \left(\frac{n}{2e}\right)^n = \infty$$

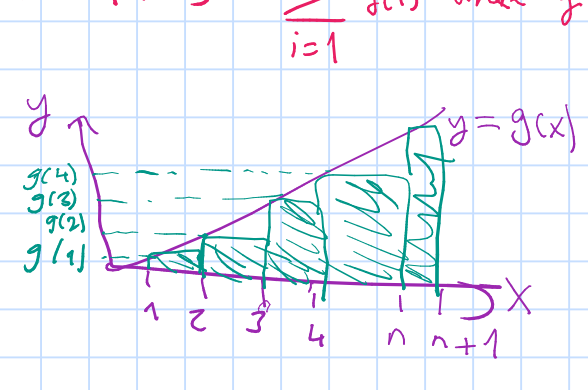
Converges to infinity

$\therefore n! \in \omega(2^n)$

$n!$ is much worse than 2^n

In analyzing an algorithm, we usually arrive at a summation Σ and then convert this summation into closed form.
Sometimes this Σ is too complex. At that time we can squeeze this summation between 2 integrals.

⇒ Let $f(n) = \sum_{i=1}^{n} g(i)$ where g is a non decreasing function.

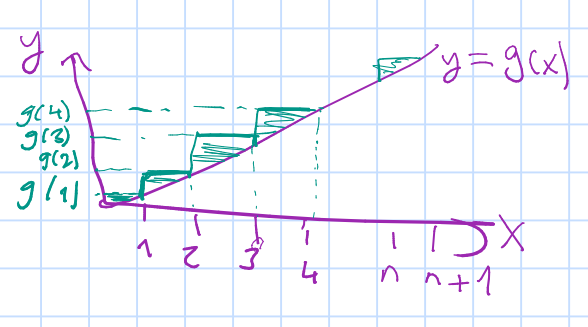


Width is always 1,

$g(1) \cdot 1 + g(2) \cdot 1 + \dots g(n) \cdot 1 = f(n)$

$\int_1^{n+1} g(x)\,dx \Rightarrow$ The entire area under the curve.

$f(n) \leq \int_1^{n+1} g(x) \cdot dx$



$g(1) \cdot 1 + g(2) \cdot 1 + \dots g(n) = \sum_{i=1}^{n} g(i) = f(n)$

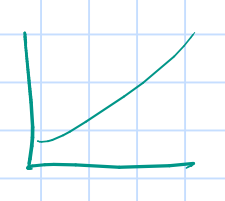
$\geq \int_0^n g(x)\,dx$

$$\int_0^n g(x)\,dx \leq \sum_{i=1}^{n} g(i) \leq \int_1^{n+1} g(x)\,dx$$

This is valid only if g is a nondecreasing function.

Ex: $f(n) = \sum_{i=1}^{n} i^2$

Non decreasing



$\int_0^n g(x)\,dx \leq f(n) \leq \int_1^{n+1} g(x)\,dx$

$\int_0^n x^2\,dx \leq f(n) \leq \int_1^{n+1} x^2\,dx$

$\frac{x^3}{3}\Big|_0^n \leq f(n) \leq \frac{x^3}{3}\Big|_1^{n+1}$

$\frac{n^3}{3} \leq f(n) \leq \frac{(n+1)^3 - 1}{3}$

$f(n) \in \Omega(n)$
$f(n) \in O(n)$

$f(n) \in \Theta(n^3)$

Ex: Sum of logarithms

$L(n) = \sum_{i=1}^{n} \log(i) = \log(n!)$

$L(n) = \underbrace{\log 1 + \log 2 + \dots + \log n}_{n \text{ terms}} \leq n \cdot \log n \in O(n \log n)$

For lower bound:

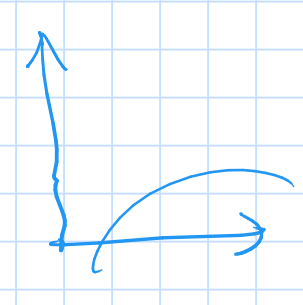
$$\underbrace{\sum_{i=1}^{\lfloor n/2 \rfloor} \log i}_{A} + \underbrace{\sum_{i=\lfloor n/2 \rfloor+1}^{n} \log(i)}_{B} \geq \underbrace{\sum_{i=\lfloor n/2 \rfloor+1}^{n} \log(i)}_{B}$$

$$= \log(\lfloor \tfrac{n}{2} \rfloor + 1) + \log(\lfloor \tfrac{n}{2} \rfloor + 1) + \log(\lfloor \tfrac{n}{2} \rfloor + 2) + \dots \log(n)$$

$n - \lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil \geq \frac{n}{2}$ terms

Each term $\geq \log(\frac{n}{2}) \in \Omega(n \log n)$

Integration Technique



$\log i$ is a non-decreasing function

$$\int_0^n \ln dx \leq L(n) < \int_1^{n+1}$$

(log or ln doesn't matter)

Derivative of $(x \ln x - x) \Rightarrow x' \ln x + x(\ln x)' - 1$
$= 1 \cdot \ln x + x \cdot \frac{1}{x} - 1$
$= \ln x$
It means true.

$$(x\ln x - x)\Big|_0^n \leq L(n) \leq (x \ln x - x)\Big|_1^{n+1}$$

$n \ln n - n - \underbrace{(0 \ln x - 0)}_{0} \leq L(n) \leq (n+1)\ln(n+1) - (n+1) - (0-1)$

$\leq (n+1)\ln(n+1) - n - 1 + 1$ (the $-1+1$ crossed out)

$L(n) \in \Theta(n \log n)$

Stirling's Formula (conjunction)

$$e^{n\ln(x) - n} \leq e^{\ln(n!)} \leq e^{(n+1)\ln(n+1) - n}$$

$$\frac{e^{n \ln x}}{e^n} \leq n! \leq \frac{e^{(n+1) \cdot \ln(n+1)}}{e^n}$$

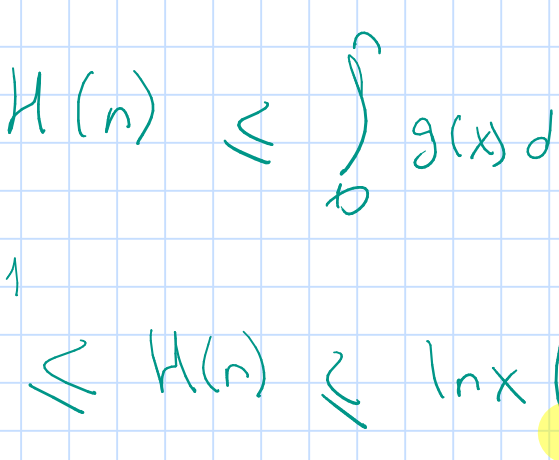
Note: $a^{\log n} = n^n$

Harmonic Series

$H(n) = \sum_{i=1}^{n} \frac{1}{i}$ (n^{th} Harmonic)

$H(1) = 1$
$H(2) = 1 + \frac{1}{2} = 1,5$
$H(3) = 1 + \frac{1}{2} + \frac{1}{3} = 2, \dots$

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



It's not non-decreasing
It's non-increasing
Now we need to change bound of integrals.

Reverse graph: is valid no

$\int_1^{n+1} g(x)\,dx \quad H(n) \leq \int_0^n g(x)\,dx$ for a non decreasing function

$\ln x \Big|^{+1} \leq H(n) \leq \ln x \Big|_0^n$ — we need to get rid of this so we make it 1 to find an upper bound. (Because $\ln 0 = -\infty$, $\ln 1 = 0$)

$\ln(n+1) \leq H(n) \leq \ln n - \underbrace{\ln(0)}_{-\infty}$

$\ln(n+1)$: Works for lower bound

∞: It means we couldn't obtain any upper bound.

$\Rightarrow H(n) = \sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \frac{1}{i}$

$\Rightarrow H(n) \leq 1 + \int_1^n g(x)\,dx$

Take first term so we can use integral bounds from 1 to n.

$\leq 1 + \ln x \Big|_1^n$

$\leq 1 + \ln n - \underbrace{\ln 1}_{0}$

$\leq \ln(n) + 1$

$\ln(n+1) \leq H(n) \leq \ln(n) + 1$

$\ln(n) \leq \ln(n+1) \leq H(n) \leq \ln(n) + 1 \leq 2\ln(n)$ (To make it more precise)

$H(n) \in \Theta(\log n)$

Week 4

Linear (Sequential) Search

Pseudo Code:

```
Function Linear Search (L[1:n], x)        (input is L & element)
    for i ← 1 to n do
        if (L[i] = x) then return (i)
        endif
    end for
    return 0
end
```

(See Python code on Learning)

A comparison based searching or sorting algorithm is an algorithm that is based on making comparisons involving list elements and then making decisions based on these comparisons.

Best case: if $x = L[1]$, then best case occurs

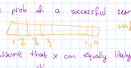
$B(n) = 1 \in O(1)$
$\Theta(1)$

Worst case: Doesn't exist or last element

If $x = L(n)$ or x doesn't exist in the list, then worst case occurs.

$W(n) = n \in \Theta(n)$

Average Case: Assume prob of a successful search is p, where $0 \le p \le 1$,



A classical assumption → Assume that x can equally likely be found in any position.
Assume elements are distinct.

Probability of i comparisons will be done, i.e. probability that x occurs in the i^{th} position.
= Prob that $x = L[i]$ = $\frac{p}{n}$ for $1 \le i \le n$
= Prob that $x = L[i]$ or prob that x is not in the list
$= \frac{p}{n} + (1-p)$ for $i = n$

$$A(n) = \sum_{i=1}^{W(n)} i \cdot p_i = \left(\sum_{i=1}^{n-1} i \cdot \frac{p}{n}\right) + n\left(\frac{p}{n} + 1 - p\right)$$
(A(n): Average case)

$$= \frac{p}{n} \sum_{i=1}^{n-1} i + p + n(1-p) = \frac{p}{n}\cdot\frac{n(n-1)}{2} + p(1-n)p$$
($\sum_{i=1}^{n-1} i$: $1+2+\dots+(n-1)$)

$$= \frac{p}{2}n - \frac{p}{2} + p + n - np = n\left(1 - \frac{p}{2}\right) + \frac{p}{2} \in \Theta(n)$$
(linear; doesn't matter what p is)

Binary Search Analysis

↳ assumes that list is sorted. Otherwise doesn't work.

Best case: $B(n) = 1 \in \Theta(1)$ if x = middle element

Worst case: It's not easy to analyze the worst case for the general n.
So firstly, if $n = 2^k - 1$ where $k \in \mathbb{Z}^+$
if x is not in the list or $x = L[1]$ or $x = L[n]$, then worst case occurs.

Compare x with $L\left[2^{k-1}\right]$, then $L\left[2^{k-2}\right]$ or $L\left[3 \cdot 2^{k-2}\right]$ ... $L[1]$ or $L[n]$

There are k comparisons (worst case comes)

$n = 2^k - 1 \Rightarrow 2^k = n+1 \Rightarrow k = \log_2(n+1)$
$W(n) = \log_2(n+1)$ for $n = 2^k - 1$
(just for particular one)

Generalizing the result as follows:

$\Rightarrow W(n) = \dots$ for all n

$2^{i-1} - 1 \le n \le 2^i - 1$

$\Rightarrow W(n)$ is an non-decreasing function.

$W(2^{i-1}-1) \le W(n) \le W(2^i - 1)$

$\log_2(2^{i-1} - 1 + 1) \le W(n) \le \log_2 2^i$

$i - 1 \le W(n) \le i$
$2^{i-1} \le n+1 \le 2^i$
$i - 1 \le \log_2(n+1) \le i$
(This method known as interpolation.)

$\log_2(n+1) - 1 \le W(n) \le \log_2(n+1) + 1$

$W(n) \in \Theta(\log n)$

Average Case: Assume prob of successful search is p ($0 \le p \le 1$)
→ it is equally likely that x can be found in any position.
→ if x isn't in the list, it is equally likely that it is on either in any one of the following $n+1$ intervals:

$x < L[1]$, $L[1] < x < L[2]$, ..., $L[n-1] < x < L[n]$, $x > L[n]$

(This tree has n internal nodes and $n+1$ leaf nodes → corresponding to $n+1$ consecutive intervals.)

Prob that x can be found in i^{th} position $L[i]$: $\frac{p}{n}$ for $1 \le i \le n$

Average number of comparisons:
If $x = L[i] \Rightarrow \frac{p}{n}$: i comparisons
If $x \ne L[i] \Rightarrow \frac{1-p}{n+1}$: i comparisons

$= \frac{p}{n} \cdot 1 + \frac{p}{n} \cdot 2 + \dots + \frac{p}{n} \cdot i + \dots + \frac{1-p}{n+1} \cdot i$
($\frac{p}{n}$: $x \in L[i]$)
$= \frac{p}{n}(1+2+3+\dots+i+\dots+n)$ → (This corresponds to the internal path length of the tree.)

Internal Path Length: The sum of the lengths of the paths from the root to the internal nodes.

Average number of comparisons $= \frac{p}{n}\left(IPL(T) + n\right)$
(Found or not $\Rightarrow$ add n)

Average number of comparisons $= \frac{1-p}{n+1} \cdot 3 + \frac{1-p}{n+1} \cdot 3 + \dots$
$= \frac{1-p}{n+1}(3 + 3 + \dots)$
(3 for this example tree.)

Leaf Path: The sum of the lengths of the paths from the root to the leaf nodes.

Average number of comparisons for unsuccessful case $= \frac{1-p}{n+1} \cdot LPL(T)$
(for unsuccessful case)

$A(n) = \sum_{x \in T_n} T(x) \cdot P(x) = \frac{p}{n}\left(IPL(T) + n\right) + \frac{1-p}{n+1} \cdot LPL(T)$

Definition: An 1-tree is a binary tree such that every node that is not a leaf has exactly 1 children.

Definition: A binary tree T is called a full binary tree of level l if T contains the max number of nodes at that level, $0 \le l \le D(T)$.
(Depth)

Definition: A 2-binary tree T is full at the second deepest level if all leaf nodes are in the levels $D(T) - 1$ and $D(T)$.

$\Rightarrow$ for a 2-tree T, $IPL(T) = LPL(T) - 2I$,
where I is the number of internal n.

For any binary tree T, $LPL(T) \ge L\lfloor \log_2 L \rfloor + 2\left(L - 2^{\lfloor \log_2 L \rfloor}\right)$,
where L is the number of leaf nodes.
(This becomes an equality for a 2-tree. & Fall at the second deepest level.)

For successful case $A(n) = \sum T(x) \cdot P(x) = \frac{p}{n}\left(IPL(T) + n\right) + \frac{1-p}{n+1}\left(IPL(T)\right)$

$= \frac{p}{n}\left(LPL(T) - n\right) + \frac{1-p}{n+1} \cdot LPL(T)$

$= \left(\frac{p}{n} + \frac{1-p}{n+1}\right) LPL(T) - p \ge \left(\frac{p}{n} + \frac{1-p}{n+1}\right)\left[(n+1)\lfloor \log_2(n+1) \rfloor + 2\left((n+1) - 2^{\lfloor \log_2(n+1) \rfloor}\right)\right] - p \ge \underbrace{\left(\frac{p}{n} + \frac{1-p}{n+1}\right)}_{\ge \frac{1}{n}} \cdot \underbrace{(n+1)\left(\lfloor \log_2(n+1) \rfloor - 1\right)}_{\ge \log_2 n}$

$\in \frown\!\!\_\!\!\frown (\log n)$ More close to the worst case than best case

$A(n) \le W(n) \in O(\log n)$
$A(n) \in \Omega(\log n)$
$A(n) \in \Theta(\log n)$

$\Rightarrow A(n) \in \Theta(\log n)$

Interpolation

We know that $f(n) \in \Theta(g(n))$ for $n = b^k (k \ge 0, b \ge 1)$,

$\Downarrow$

$\left[f(b^k) \in \Theta(g(b^k)) \text{ for } n \in \mathbb{N}\right]$ for all n

We want to show that $f(n) \in \Theta(g(n))$ for all $n \in N$

Θ - Invariant Understanding

A function $h(n)$ is called eventually non-decreasing if there exists an integer n_0 such that $h(n) \le h(m)$ whenever $m \ge n_0 \ge 0$.

For a function $h(n)$, if $h(c \cdot n) \in \Theta(h(n))$ for any positive constant c, then $h(n)$ is called Θ-invariant under scaling, and $\Theta(h(n))$ is called a scale invariant class.

Ex: $h(n) = n^2 \in \Theta(n^2)$

$h(c \cdot n) = \underbrace{c^2}_{\text{Const}} \cdot n^2 \in \Theta(n^2)$

Ex: $h(n) = \log_2(n) \in \Theta(\log n)$
$h(cn) = \log_2(c \cdot n) \in \Theta(\log n)$
$= \underbrace{\log_2 c}_{\text{Const}} + \log n$

Week 5

$h(n) = 2^n \in \Theta(2^n)$
$h(cn) = 2^{cn}$
$= (2^c)^n \notin \Theta(2^n)$
$\in \Theta(2^{cn})$

* Exponential functions are not Θ invariant.

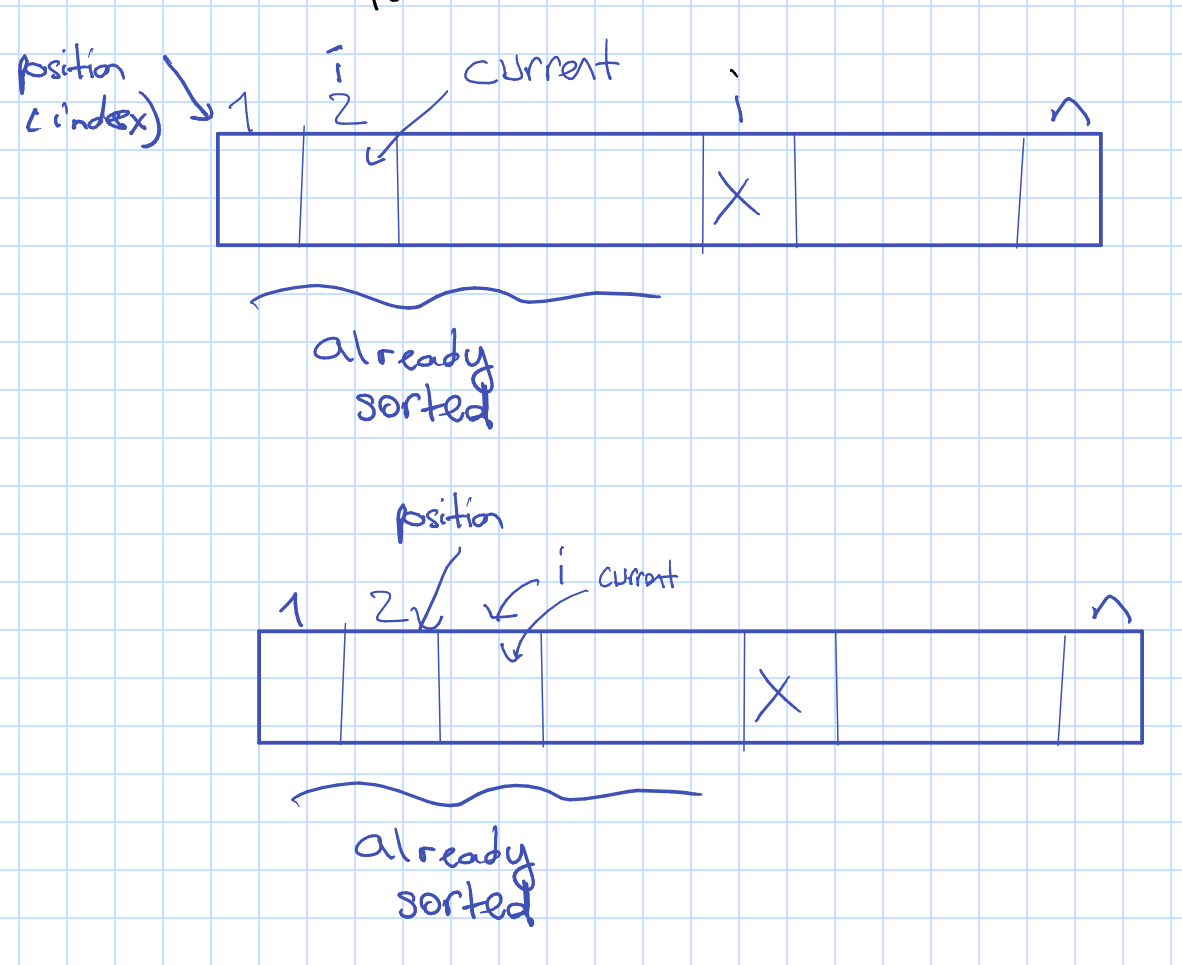
* Suppose that $f(b^n) \in \Theta(g(b^n))$ for $b>1$ and $n \in \mathbb{N}$. Suppose that $f(n)$ and $g(n)$ are eventually non-decreasing and $g(n)$ is Θ-invariant under scaling. Then $f(n) \in \Theta(g(n))$ $\forall n \in \mathbb{N}$.

Ex: $W(n) \in \Theta(\log n)$ for $n = 2^k - 1$
$\log(n+1) - 1 < W(n) < \log_2(n+1) + 1$

$W(n)$ is non-decreasing.
$\log n$ is non-decreasing.
$\log n$ is Θ invariant.
⇓
$W(n) \in \Theta(\log n)$ $\forall n \in \mathbb{N}$

Insertion Sort → incremental, we need non-constant additional memory

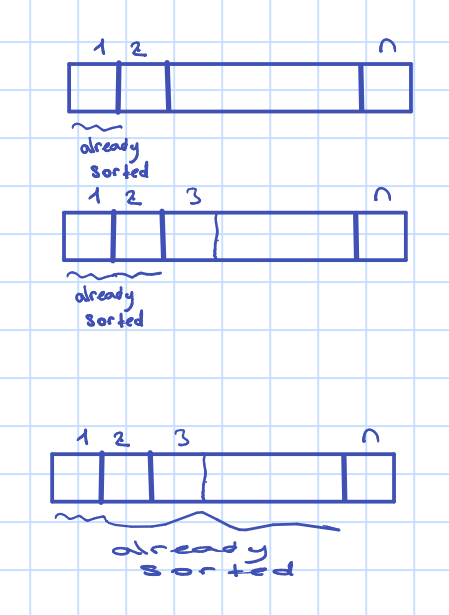
```
procedure insertionSort (L[1:n])
  for i=2 to n do
    current = L[i]
    position = i-1
    while((position ≥ 1) and current < L[position]) do
      L[position+1] = L[position]
        position = position - 1
    end while
      L[position+1] = current
  end for
```



Best case: Don't enter to while loop.
The basic operation executes once and the while loop terminates.
This occurs if the input is already sorted.
Complexity: $B(n) = \sum_{i=1}^{n} 1 = n-1 \in \Theta(n)$
↳ for loop always executes.

Worst case: For each iteration of the for loop, the basic operation is executed max number of times.
This situation occurs if the list $L[1:n]$ in reverse order.
So worst case complexity is $W(n) = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

Average Case:



Let T_i = # of basic operations at step i for $1 \le i \le n-1$

$T = T_1 + T_2 + \dots + T_{n-1} = \sum_{i=1}^{n-1} T_i$

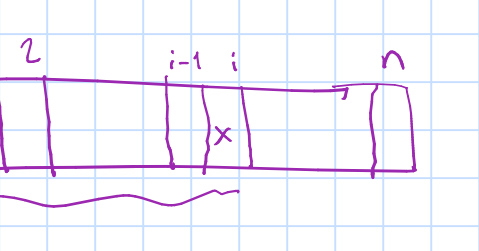
$A(n) = E[T] = E\left[\sum_{i=1}^{n-1} T_i\right] = \sum_{i=1}^{n-1} E[T_i]$

$= E[T_1] + E[T_2] + \dots + E[T_{n-1}]$

Calculate $E[T_i]$:

$E[T_i] = \sum_{j=1}^{i} j \cdot P(T_i = j)$

$E[T_i]$ ↓ Random variable; $P(T_i=j)$: Probability that there are j comparisons in the i^{th} step.



1 comparison will occur if $x = L[i] > L[i-1]$
2 comparisons will occur if $x = L[i-2] < x < L[i-1]$
3 comparisons will occur if $x = L[i-3] < x < L[i-2]$
i comparisons will occur if $x = L[1] < x < L[2]$
i comparisons will occur if $x < L[1]$
↳ Smallest of new already sorted list.

* We assume that each of these intervals is equally likely.
* There are $i+1$ possible cases

$$P(T_i = j) = \begin{cases} \frac{1}{i+1} & \text{if } 1 \le j \le i-1 \\ \frac{2}{i+1} & \text{if } j = i \end{cases}$$

Expected value of T_i

$E[T_i] = \sum_{j=1}^{i-1} j \cdot \frac{1}{i+1} + i \cdot \frac{2}{i+1}$

$= \frac{i(i-1)}{2(i+1)} + \frac{2i}{i+1}$ (2)

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

$= \frac{i^2+3i}{2(i+1)}$

$A(n) = E[T] = \sum_{i=1}^{n-1} E[T_i]$

$\sum_{i=1}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n(n-1)}{4} + n - 1 - \sum_{i=1}^{n-1}\frac{1}{i+1}$

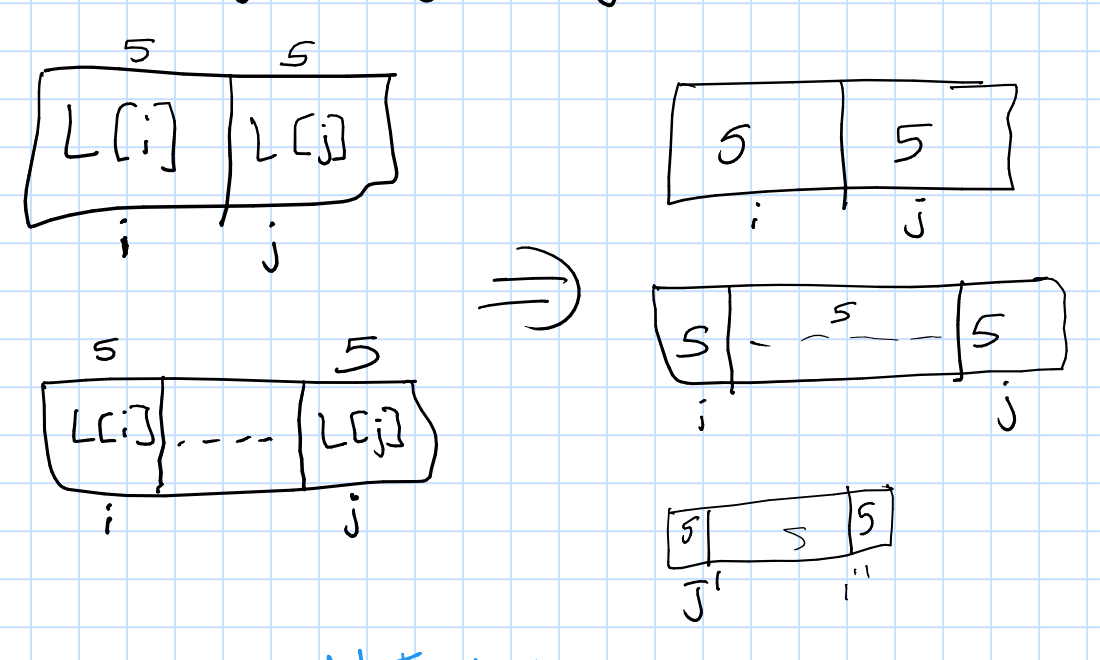
$= \frac{n(n-1)}{4} + n - H(n) \in \Theta(n^2)$

$\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = H(n) - 1$

Definition: A sorting algorithm is an in-place (yerinde, uygun) sorting algorithm if only a constant amount of memory is used in addition to that needed for the input list. (quick sort ✓, merge sort X)

Definition: A sorting algorithm is an online sorting algorithm if the entire list is not input to the algorithm in advance, but instead elements are added to the list over time. The algorithm maintains the dynamic list in sorted order.
Insertion sort is an online algorithm.
Quick sort is NOT an online algorithm.
Merge sort is NOT an online algorithm.

Definition: A sorting algorithm is a stable sorting algorithm if given any 2 elements $L[i], L[j]$ where $i<j$ and $L[i] = L[j]$, the final positions i', j' of $L[i], L[j]$ respectively satisfy $i' < j'$



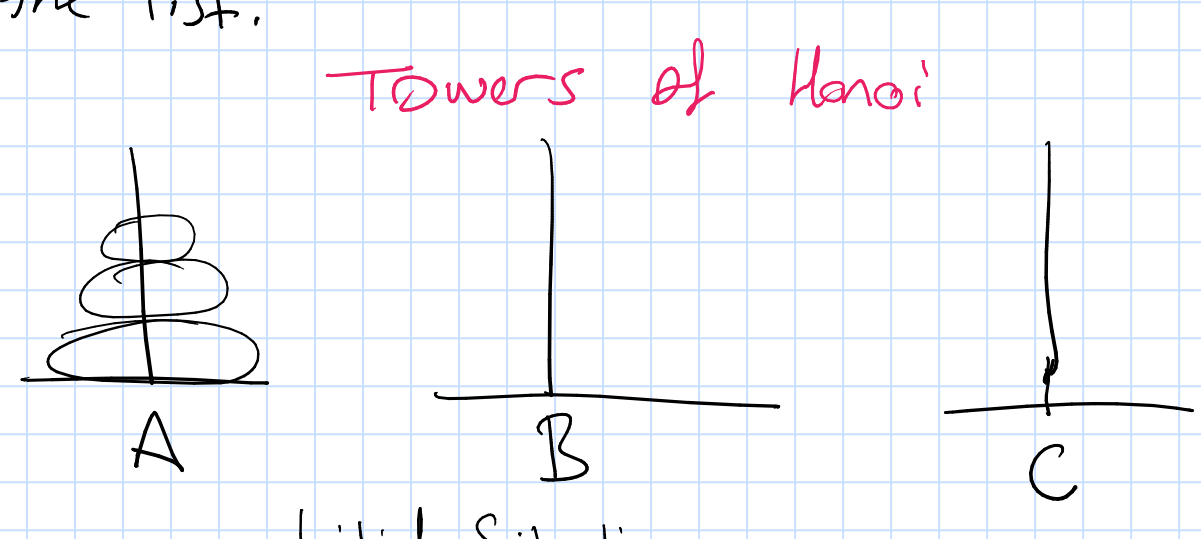
Quicksort is NOT stable.

Proposition: Given integers n, k, where $k \le n$, suppose that $L[1,n]$ is a list such that every element in the list is no more than k positions from it's stable final position in the sorted list L.
↳ The unique final position of the element after any stable sorting algorithm is applied.

Then insertion sort performs at most $2k(n-1)$ comparisons when sorting the list.

Towers of Hanoi

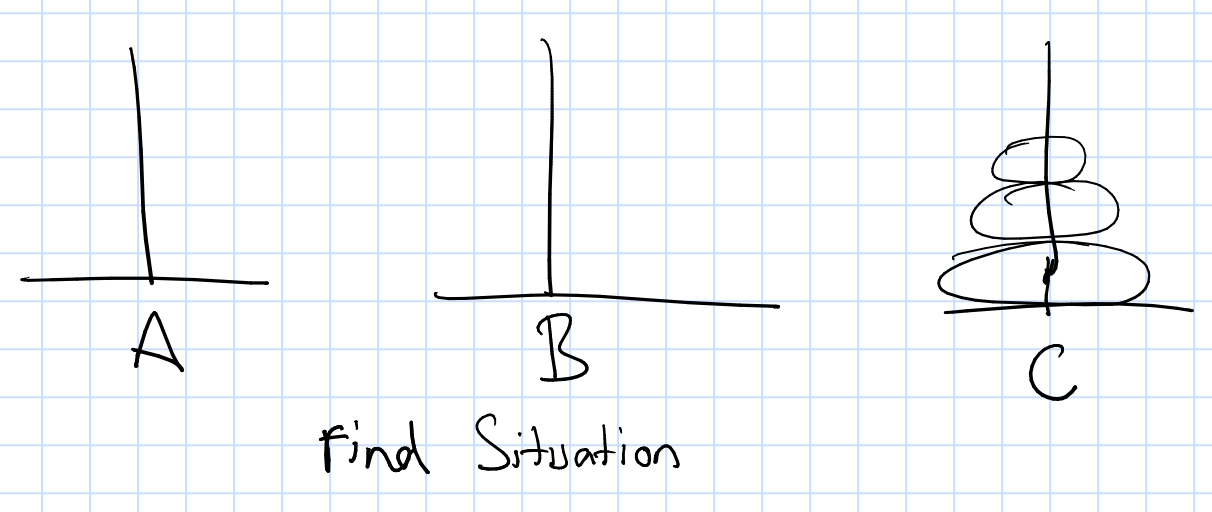
n = # of disks



Initial Situation

Rule 1: At each step, we can only move only the pack on top of the other packs. We can only move 1 pack at a time.
Rule 2: We cannot place a larger disk on top of a smaller one.

Final situation: Goal of the situation



Final Situation

$T(n)$ = Complexity of Towers for an input of size n.
$T(n) = 2T(n-1) + 1$ for $n > 1$
↓
$T(1) = 1$
$T(2) = 2 \cdot T(1) + 1 = 2 \cdot 1 + 1 = 3$
$T(3) = 2 \cdot T(2) + 1 = 2(2T(1)+1)+1 = 2^2 \cdot T(1) + 2 \cdot 1 + 1$
$T(4) = 2 \cdot T(3) + 1 = 2^3 T(2) + 2^2 + 2 + 1 = 2T(3)$
$T(n) = 2^{n-1} \cdot T(1) + 2^{n-2} + 2^{n-1} + \dots + 2^1 + 2^0$

```
Procedure Towers(A,B,C,n)
  if n=1 then
    write("Move from A to C")
  else
    call Towers(A,C,B,n-1) // Move n-1 disks from A to B
    write("Move from A to C"  // Move the largest disk from A to C
    call Towers(B,A,C,n-1)  // Move n-1 disks from B to C
```

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

$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x-1}$ for $x \in \mathbb{R}^+$, $x \ne 1$, $n > 0$, $n \in \mathbb{Z}^+$

$\sum_{k=0}^{n-1} a^n = a^0 + a^1 + \dots a^{n-1} = \frac{a^n - 1}{a-1}$ $a = 2 > 1$

General form of Recurrence Relation

$X(n) = C_1 \cdot X(n-1) + C_2 X(n-2) + \dots + C_k \cdot X(n-k) + f(n)$

$X(0), \dots, X(k-1)$ are given. (initial conditions)

OR

$X(n) = c \cdot X\left(\frac{n}{b}\right) + f(n)$ where $X(1)$ is given.

k^{th} order recurrence relation, k is usually 1 or 2.

$c_1, c_2 \dots$ ⇒ denotes the # of recursive calls

$f(n)$'s denote the extra operations in the algorithms.

Forward Substitution

Ex: $X(n) = 2 \cdot X(n-1) + 1$, $n > 1$, $X(1) = 1$

$X(1) = 1$
$X(2) = 3$
$X(3) = 7$
$X(4) = 15$
$X(5) = 31$

We try to observe a pattern.
Guess $X(n) = 2n \cdot x$
$X(n-1) = 2(n-1)$
$2n \stackrel{?}{=} 4(n-1) + 1$
$2n = 4n - 4 + 1$
$2n = 4n - 3$ $3 = 2n$, $n = \frac{3}{2}$

Just a guess Not a proof ↳ To make it proof ↓

Guess $X(n) = 2^n - 1$
$X(n-1) = 2^{n-1} - 1$
$X(n) = 2X(n-1) + 1$
$2^n - 1 \stackrel{?}{=} 2(2^{n-1} - 1) + 1$
$2^n - 2 + 1 = 2^n - 1$

→ Solve the new recurrence
$y(n) = y(0) + \sum_{i=1}^{n} d_i$
→ Return to the original variable
$X(n) = b_1 \cdot b_2 \dots b_n \left(y_0 + \sum_{i=1}^{n} d_i\right)$

Backward Substitution

Ex: $X(n+1) = b_{n+1} \cdot X(n) + c_{n+1}$; $n \ge 0$, $x(0)$ is given

$X(n+1) = b_{n+1} \dots b_1 X(0) + b_{n+1} \dots b_2 c_1 + b_{n+1} \dots b_3 c_2 + \dots + c_{n+1}$

We can apply a trick to the original formula to obtain another formula.

→ Make a new change of variable that leads to a new recurrence

$X(n) = b_1 \cdot b_2 \dots b_n y(n)$; $n \ge 1$; $y(0) = X(0)$

$(b_1 \dots b_{n+1}) y(n+1) = (b_{n+1} \cdot b_1 \dots b_n) \cdot y_n + c_{n+1}$

We make the coefficients the same, so that we can eliminate them.

Now, this can be easily solved by backward substitution.

$y(n+1) = y(n) + d_{n+1}$ where $d_{n+1} = \frac{c_{n+1}}{b_1 \dots b_{n+1}}$
$y(0)$ is given.

Ex: $X(n+1) = 3 \cdot X(n) + n$; $n \ge 0$, $X(0) = 0$

$X(n) = 3^n \cdot y(n)$

(*) $3^{n+1} \cdot y(n+1) = 3^{n+1} \cdot y(n) + n$; $y(0) = 0$, $n \ge 0$

$y(n+1) = y(n) + \frac{n}{3^{n+1}}$ ⇒ $y(n) = \sum_{i=1}^{n-1} \frac{i}{3^{i+1}}$ (Guess)

$X(n) = 3^n \cdot \sum_{i=1}^{n-1} \frac{i}{3^{i+1}}$
↳ if we cannot solve this summation part exactly, we can squeeze it between 2 integrals and find the Θ solution.

$X(n+1) = a \cdot X(n) + b \cdot X(n-1)$; $n \ge 1$, $X(0)$, $X(1)$ are given.
↓
2^{nd} order recurrence relation.

$\alpha^2 = a\alpha + b$ ⇒ Characteristic equation of this recurrence.
⇓
Solve this equation to find the roots for α and then we express the solution of the recurrence in terms of these roots.

$\alpha^{n+1} = a \cdot \alpha^n + b \cdot \alpha^{n-1}$
Divide all by α^{n-1}
$\alpha^2 = a\alpha + b$ Roots α^+ and α^-

i) if both α^+ and α^- are real and distinct, then $X(n) = c_1 \cdot \alpha_+^n + c_2 \cdot \alpha_-^n$ for some c_1, c_2
↳ we can find these constants from the initial conditions.

ii) if $\alpha_+ = \alpha_-$ (say, α) and it is real, then
$X(n) = c_1 \cdot \alpha^n + c_2 \cdot n \cdot \alpha^n$

iii) If the roots are complex, it won't occur in our algorithm analysis

Ex: Recurrence for Fibonacci Numbers
$f(n+1) = f(n) + f(n-1)$; $n > 1$, $f(0) = 0$, $f(1) = 1$
Characteristic equation: $\alpha^2 = \alpha + 1$ (BAK)

$\alpha^2 - \alpha - 1 = 0$ Roots: $\alpha_+ = \frac{1+\sqrt{5}}{2}$ $\alpha_- = \frac{1-\sqrt{5}}{2}$

$f(n) = c_1 \cdot \alpha_+^n + c_2 \alpha_-^n = c_1\left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n$

$f(0) = 0$ ⇒ $c_1 + c_2 = 0$
$f(1) = 1$ ⇒ $c_1 \cdot \left(\frac{1+\sqrt{5}}{2}\right) + c_2 \cdot \left(\frac{1-\sqrt{5}}{2}\right) = 1$
$c_1 + c_2 \cdot \sqrt{5} + c_2 - c_2\sqrt{5} = 2$

$\underbrace{c_1 + c_2}_{0} + c_1 \cdot \sqrt{5} - c_2\sqrt{5} = 2$
$c_1 - c_2 = \frac{2\sqrt{5}}{5}$
$+ \; c_1 + c_2 = 0$
$c_1 = \frac{\sqrt{5}}{5}$
$c_2 = \frac{-\sqrt{5}}{5}$

$f(n) = \frac{\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n$

Ex: $X(n+1) = 2 \cdot X(n) - X(n-1)$, $n \ge 1$, $X(0) = 1$, $X(1) = 5$

$\alpha^2 = 2\alpha - 1$ $(\alpha - 1)^2 = 0$
$\alpha^2 - 2\alpha + 1 = 0$ ↓ $\alpha = 1$

$X(n) = c_1 \cdot \alpha^n + c_2 \cdot n \cdot \alpha^n$
$X(n) = c_1 + c_2 \cdot n$
$X(0) = c_1 = 1$
$X(1) = c_1 + c_2 = 5$
↳ 4
$X(n) = 4n + 1$

Ex:

① One type of recurrence ⇒ $X(n) = a \cdot X(n-1) + f(n)$, $X(1) = b$
⇓
Decrease by 1 algorithms (like insertion sort)

② $X(n) = a \cdot X\left(\frac{n}{b}\right) + f(n)$; where $X(1) = c$, $b > 1$
b is usually 2
if $a = 1$ ⇒ Decrease-by-a-constant algorithms → Ex: Binary search
if $a > 1$ ⇒ Divide and conquer algorithms → Ex: Merge sort, Quick sort

Week 6

Ex:

(1) $X(n) = a^{n-1} \cdot b + \sum_{i=0}^{n-1} a^{n-i} \cdot f(i)$ (By backward substitution)

⇒ If $a=1$ and $f(n)=1 \Rightarrow X(n) = b + \sum_{i=0}^{n-1} 1 = b + (n-1) \in \Theta(n)$

⇒ If $a=1$ and $f(n) = \log n \Rightarrow X(n) = b + \sum_{i=0}^{n-1} \log i$

$= b + \log(n!) \in \Theta(n \log n)$

(2) if n is not a power of b, we'll obtain something like $X(n) = a \cdot X\left(\frac{n}{b}\right) + \dots$

Assume that $n = b^k \Rightarrow X(n) = a^k \cdot c + \sum_{i=0}^{k-1} a^i \cdot f\left\lceil \frac{n}{b^i} \right\rceil$

(it depends on this function.)

If $f(n) = d \Rightarrow X(n) = a^k \cdot c + d \cdot \sum_{i=0}^{k-1} a^i$

$a^k = a^{\log_b n} \Rightarrow \left(b^{\log_b a}\right)^{\log_b n}$

$a^k = n^{\log_b a} \Rightarrow X(n) = n^{\log_b a} \cdot c + d \cdot \left(\frac{n^{\log_b a} - 1}{a-1}\right)$

$= n^{\log_b a} \left(c + \frac{d}{a-1}\right) - \frac{d}{a-1}$

$\in \Theta(n^{\log_b a})$

if $a=b \Rightarrow n^{\log_b a} \cdot c + d \cdot n \cdot \sum_{i=1}^{k-1} 1 = cn + dn$

$= cn + dn \cdot \log_b n$

$\in \Theta(n \log n)$ (this is the case for merge sort)

We have obtained these results only for $n = b^k$

However, since all of these functions are Θ invariant and non-decreasing, we can generalize them for all n.

Master Theorem

Let $X(n)$ be an eventually non-decreasing function that satisfies the recurrence relation:

$X(n) = a \cdot X\left(\frac{n}{b}\right) + f(n)$, where $n = b^k (k = 1, 2, \dots)$

and $X(1) = c$, where $a \geq 1$, $b \geq 2$, $c > 0$

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ then

$$X(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \cdot \log n) & \text{if } a = b^d \ \forall n \\ \Theta\left(n^{\log_b a}\right) & \text{if } a > b^d \end{cases}$$

Abbiegia k Dicom $\exists d$

Ex: merge sort

$X(n) = 2X\left(\frac{n}{2}\right) + \underbrace{f(n)}_{\Theta(n)}$

$a = 2$

$b = 2$

$f(n) \in \Theta(n)$

$d = 1$ $\quad b^d = 2^1 = 2 = a$

$\Theta(n \log n)$

Ex: Towers of Hanoi

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

$a = 2$

$b = 2$

$d = 0$

We cannot apply.

QUICKSORT

Hoare invented in 1960:

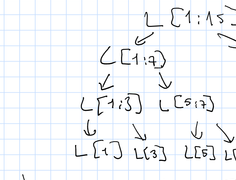
```
procedure Quicksort(L[low:high])
    if high > low then
        call Rearrange(L[low:high], position)
        call Quicksort(L[low:position-1])
        call Quicksort(L[position+1:high])
    end if
```


```
procedure rearrange(L[low:high], position)
    right = low
    left = high+1
    X = L[low]
    while right < left do
        repeat right = right+1 until L[right] ≥ X
        repeat left = left-1 until L[left] ≤ X
        if right < left then
            call interchange(L[left], L[right])
```

Basic Operation: $L[right] \geq X$, $L[left] \leq X$ → comparison between list elements

Rearrange module always performs high − low + 2 operations.

Best Case: divide 2 equal sized sublist each time, it's best case.



In rearrange, always high − low + 2 comparisons will be performed.

The tree has k levels.

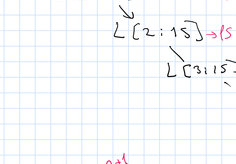
In $k-1$ of these levels, these are $n+1$ comparisons.

$B(n) = (k-1)(n+1) = \left(\log_2(n+1) - 1\right)(n+1) \in \Omega(n \log n)$ for $n = 2^k - 1$

The functions are non-decreasing and Θ invariant. Hence we can generalize this result for all n.

Worst Case:

For quicksort, if the list is already sorted, this is the worst case.



$W(n) = \sum_{i=3}^{n+1} i = \Theta(n^2)$

Average Case: Not in exam.

WEEK 7-8

Quicksort Average Case Analysis

Assume that it is equally likely that the pivot element $L[low]$ will be placed in any position after rearrange.
(The proper position of the pivot is the important thing in this algorithm.

$$T = T_1 + T_2$$

T: Random variable

T_1: # of operations in rearrange()

T_2: # of operations in recursive calls

$$A(n) = E[T] = \underbrace{E[T_1]}_{\text{fixed } (n+1)} + \underbrace{E[T_2]}_{\text{Depends on where the pivot has been placed.}}$$

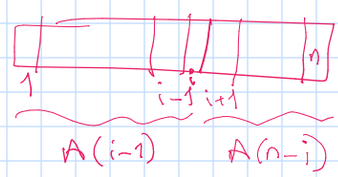
If the proper position of the pivot was $L[1]$, then 2 sublists are
$L[1:0]$ and $L[2:n]$

If the proper position of the pivot was $L[2]$, then 2 sublists are
$L[1:1]$ and $L[3:n]$

If the proper position of the pivot was $L[i]$, then 2 sublists are
$L[1:i-1]$ and $L[i+1:n]$

$$E[T_2] = \sum_{x} E[T_2 / X = x] \cdot \underbrace{P(X = x)}_{\frac{1}{n}}$$

x: position of the pivot



$$A(n) = \underbrace{(n+1)}_{E[T_1]} + \sum_{i=1}^{n} E[T_2 / X = i] \cdot \frac{1}{n}$$

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

Full history recurrence relation

$i=1 \Rightarrow \cancel{A(0)} + \cancel{A(n-1)}$
$i=2 \Rightarrow \cancel{A(1)} + \cancel{A(n-2)}$
$i=3 \Rightarrow \cancel{A(2)} + \cancel{A(n-3)}$
$\vdots$
$i=n-1 \Rightarrow \cancel{A(n-2)} + \cancel{A(1)}$
$i=n \Rightarrow \cancel{A(n-1)} + \cancel{A(0)}$
$+$

$$(n+1) + \frac{2}{n}\left[A(0) + A(1) + \dots A(n-1)\right]$$

Full history recurrence relation

$$n \cdot A(n) = n(n+1) + 2\left[A(0) + \dots + A(n-1)\right]$$

$$(n-1)A(n-1) = n(n-1) + 2\left[A(0) + \dots + A(n-2)\right]$$

$$n \cdot A(n) - (n-1) \cdot A(n-1) = 2n + 2A(n-1)$$

↳ Divide by $\frac{1}{n(n+1)}$

$$*\ \boxed{\frac{A(n)}{n+1} - \frac{A(n-1)}{n} = \frac{2}{n+1}}$$ BAK

Change of variable

$t(n) = \frac{A(n)}{n+1}$ $\Rightarrow t(n) = t(n-1) + \frac{2}{n+1}$ $\Rightarrow$ Now, it's just first order recurrence relation.

$A(0) = 0$

$t(0) = \frac{A(0)}{1} = 0$

by backward substution

$$t(n) = \sum_{i=2}^{n} \frac{2}{i+1} = 2\,\underbrace{H(n+1)}_{\text{Harmonic series}} - 3$$

$$A(n) = t(n) \cdot (n+1) = 2(n+1)\underbrace{H(n+1)}_{\log(n+1)} - 3(n+1) \quad \in \Theta(n \log n)$$

$\Rightarrow$ Average and best case complexity of quicksort are very closed to each other.

$\Rightarrow$ more detailed analysis shows that $A(n) \cong 1.38\, B(n)$

GRAPH SEARCH AND TRAVERSAL ALGORITHMS

DEPTH FIRST SEARCH (DFS) and BREADTH FIRST SEARCH (BFS)

```
procedure DFS (G,V)
    call visit(v)
    mark[v] = 1   # Visited
    u = v
    call next(u,w,found)  # Finds a unvisited neighbor
    while found or not(Empty(Stack))
        if found then
            call push(Stack,u)
            Mark[w] = 1
            call visit(w)
            u = w
        else
            call Pop(stack,u)
        end if
        call Next(u,w,found)
```

Ex: Bipartite Graphs (Graph Map coloring, planar graph)

```
Procedure BFS(G,V)
    call visit(v)
    Mark[v] = 1
    Enqueue(Queue,v)
    while not(Empty(Queue)) do
        Dequeu(Queue,u)
        for each vertex w adjacent to u, do
            if mark[w] = 0 then
                call Enqueue(Queue,w)
                Mark[w] = 1
                Visit(w)
            end if
        end for
    end while
end
```

P2-ALGORITHM DESIGNING

Brute force and Exhaustive Search

Exhaustive Search:

It's an important special case of brute force (easiest strategy)

Ex: Exponentiation

Recall: Computing $a^n \bmod m$ for large integers is an principal component of a leading encryption algorithm.

$$a^n = \underbrace{a+a+a+a \ldots + a}_{n \text{ times}}$$

BF is applicable to a wide variety of problems, Ex: Selection sort, Bubble sort

Selection Sort

Scan the entire list to find its smallest element.
Exchange it with the first element.
Put the smallest element in its find position in the sorted list.
Then continue with finding the second smallest etc [Recursively]
If the list has size n, the list is sorted after $n-1$ passes.

Pseudocode for Selection Sort.

```
Algorithm SelectionSort(A[0 ..... n-1])
    for i ← 0 to n-2 do
        min ← i
        for j ← i+1 to n-1 do
            if A[j] < A[min]
                min ← j
            end if
            swap A[i] and A[min]
        end for
    end for
```

Analysis:

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

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

$$= (n-1) + (n-2) + \ldots + 1$$

$$= \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$$

Bubble Sort

→ Compare adjacent elements
→ Exchange them if they are out of order

⇒ By doing this repetetively, we end up "bubbling up" the largest element to the last position on the list.

→ The next pass bubbles up the second largest etc.

→ Until after $n-1$ passes the list is sorted.

Observation:

If a pass through the list makes no exchanges, then the list has been sorted and we can stop the algorithm.

↳ This is a new version of the algorithm but it's sorting time is still $\Theta(n^2)$

Pseudocode for Bubble Sort (A[0 ..... n-1])

```
for i ← 0 to n-2 do
    for j ← 0 to n-2-i do
        if A[j+1] < A[j]
            swap A[j] and A[j+1]
        end if
    end for
end for
```

Ex:

28	2	10	5	11	20
2	28 ↔	10	5	11	20
2	10	28 ↔	5	11	20
2	10	5	28 ↔	11	20
2	10	5	11	28 ↔	20
2	10	5	11	20	28

Analysis:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

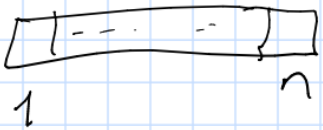
$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= (n-1) + (n-2) + \dots + 1$$

$$= \frac{(n-1)n}{2} \in \Theta(n^2)$$

Sequential Search and Brute force String Matching

String Matching Problem: Given a string of n characters called the "text" and a string of m characters called the "pattern", find a substring of the text that matches the pattern.



Ex: NOBODY_NOTICED_HER Find Pattern NOT.

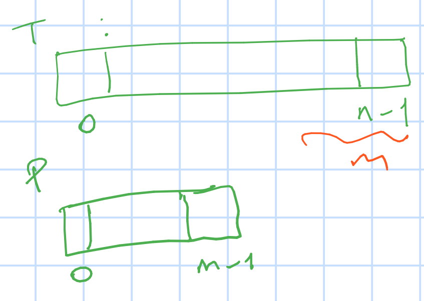
Possible application: DNA sequence alignment.

Week 8 (Actually 10)

Brute force String Matching Algorithm

Psuedocode BFStringMatch (T[0.....n-1], P[0.......m-1])

```
for i←0 to n-m do
    j←0
    while j<m and P[j] = T[i+j] do
        j←j+1
    end while
    if j=m
        return i
    end if
end for
return -1
```

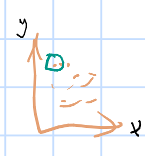


Analysis:

NONONONONOT ⇒ Worst case

The algorithm may have to make all m comparisons before shifting the pattern and this can happen for each of the n-m+1 tries.
Then in the worst case, the algorithm makes $m(n-m+1)$ character comparisons.
Complexity: $O(nm)$

Closest Pair and Convex Hull Problems By Brute Force

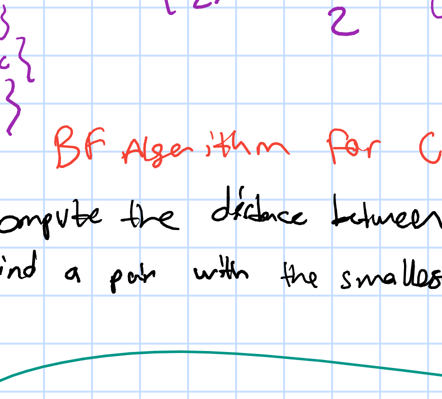


These problems arise in computational geometry and operations research.

Closest Pair Problem: finding the 2 closest points in a set of points.

Points can represent airplanes, post offices, DB records etc.
↳ 2 closest planes are the most probable collusion candidates.

Another app is cluster analysis in statistics.
↳ Similarity metric is used for Euclidian Distance. (Hamming Distance)



For simplicity we consider the 2d case.
Points P_i (x_i, y_i)
Distance btwn 2 points.
P_i and P_j $= d(P_i, P_j) \Rightarrow \sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$

Q: How many # subsets in n sized set,
$\{a, b, c, ...\}$
$\{a, b\}$
$\{b, c\}$
$\{a, c\}$
$\binom{n}{2} = \frac{n(n-1)}{2}$ (Entire search space)

BF Algorithm for CP:
- Compute the distance between each pair of points.
- find a pair with the smallest distance.

Psuedocode

```
d ← ∞
for i←1 to n-1 do
    for j←i+1 to n do
        d ← min(d, sqrt((xi - xj)^2 + (yi - yj)^2))
    end for
end for
return d
```

(Basic operation: $sqrt((x_i-x_j)^2+(y_i-y_j)^2)$)

Even today, sqrt computation is costly, mainly because it involves dealing with irrational number. Can it be avoided? YES

Only compare $(x_i-x_j)^2+(y_i-y_j)^2$ since sqrt is strictly increasing.
So the base operation will be squaring a number.

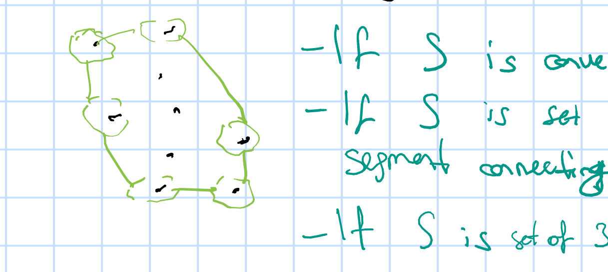
$$C(n) = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1}(n-i) = (n-1)+(n-2)+\dots+1$$

(inner sum: n-i time)

$$= \frac{n(n-1)}{2} \in \Theta(n^2)$$

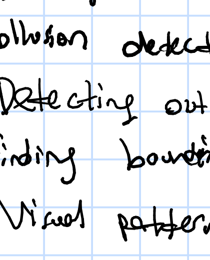
Convex Hull Problem

Convex = Set of Points (finite or infinite) in the plane is called convex (↗ distinction) if for any 2 points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set.



To be convex, all lines should be convex.

Convex Hull: The convex hull of a set S of points is the smallest convex set containing S. The smallest requirement means that the convex hull of S must be a subset of any convex set containing S.



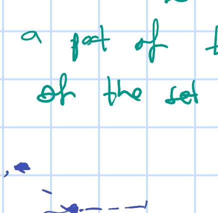
- If S is convex, it's convex hull is obviously itself.
- If S is set of 2 points, it's convex hull is the line segment connecting them.
- If S is set of 3 points that are not on the same line, its convex hull is the triangle with vertices of the given points.

Applications: Convex hull provides convenient approximations of object shapes and data sets given.
In computer animations, replacing objects by their convex hull speeds up collusion detection.
→ Detecting outliers by some statistical techniques.
– Finding bounding boxes (MS office Visio)
– Visual pattern matching (Detecting license plates)

Theorem: The convex hull of any set S of n>2 points not all on the same line is a convex polygon with the vertices at some of the points of S.
↳ we call them "extreme points"

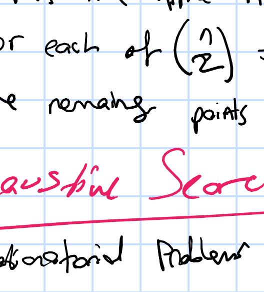
⇒ we need to find the points that will serve as the extreme points.

Extreme Point: of a convex set is a point of this set that is not middle point of any line segment with endpoints in the set.



→ we also need to know which pairs of points need to be connected to form the boundary of the convex hull.
We can address this issue by listing the extreme points in a clockwise or counter-clockwise order.

Observation: A line segment connecting 2 points P_i and P_j of a set of n points is a part of the convex hulls boundary if and only if all the other points of the set lie on the same side of the straight line through these points.



n Choose 2 pairs

For each pair, we need to make linear search for n-2 other points,
At the end it will be n^3
(This is brute force)
Exhaustive search ⇒ generating all will be expo

Algorithm
Repeating this test for every pair of points yields a list of line segments that make up the convex hulls boundary.
(*) What is the time efficiency of this algorithm?
For each of $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of distinct points, we need to check whether all of the remaining points are on the same side. ⇒ $O(n^3)$ Brute force

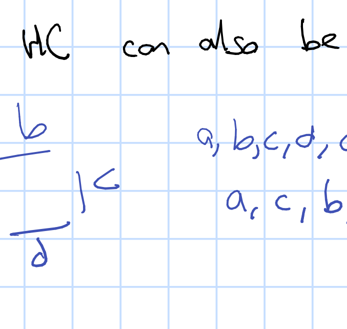
Exhaustive Search (ES)

Combinatorial Problems deal with combinatorial objects such as permutations, combinations and subsets of a given set.

Many of them are optimization problems.
→ ES is simply a BF approach to combinatorial problems.
It suggests generating each and every element of the problem domain.

Traveling Salesman Problem (TSP)

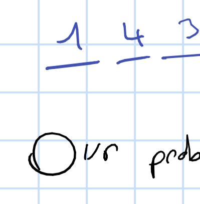
↳ To find the shortest (meaning: min-cost) tour through a given set of cities that visits each city exactly once and returns to the original city. (Min cost Hamiltonian Cycle)



a, b, c, d, e
a, c, e, b, d, a

An ES approach to TSP: Generate all Hamiltonian Cycles and choose the one with min-cost.

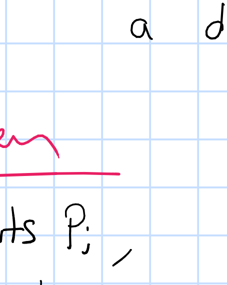
A HC can also be defined as a sequence of n+1 adjacent vertices $(v_n, v_{n+1}, \dots v_{(n+1)n}, v$



a, b, c, d, a
a, c, b, d, a

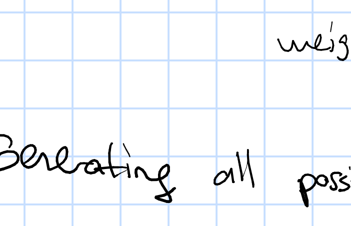
a, b, c, d, a
a, b, d, c, a
a, c, b, d, a
a, c, d, b, a
⋮

5 2 3 4 1
n n-1 n-2 -- --
n!
1 4 3 2 5



→ (n-1)!

Our problem is like ordering around a ring.



a b c d a
a c b d a
a d b c a

$\frac{(4-1)!}{2}$ = 3 possibilities (enumeration)

Knapsack Problem

n items, profits P_i,
weights w_i,
weight limit W

The goal is to put the items in the knapsack in such a way that the weigh limit is not exceeded and total profit (value) is maximized.

Generating all possible subsets (2^n Subsets) is the ES approach.

Dijkstra basically a dynamic programming that eliminates unnecessary to find shortest path.

Both TSP and Knapsack are NP-hard problems.

Assignment Problem

n people, n jobs, 1 person/job
c[i,j]: Cost of assigning person i to job j.

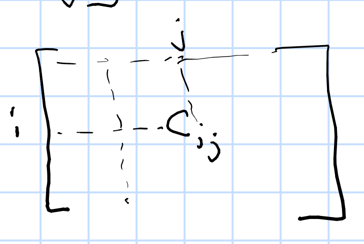
WEEK 9 → 11

Assignment Problem

n people, n jobs, 1 person/job

$c[i,j]$: Cost of assigning person i to job j.

$C[i,j]$ = Cost of assigning person i to job j.

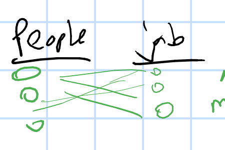


The problem boils down to selecting one element in each row of the matrix so that all selected elements are in different columns and total sum of the selected elements are in different columns and the total sum of the selected elements is the smallest possible.

⇒ No obvious solutions seems to work here.
Thus, opting for exhaustive search may appear as an unavoidable evil.

→ We can describe all feasible solutions to the assignment problem as n-tuples $(j_1, j_2, j_3, \ldots j_n)$ in which the i^{th} component indicates the column of the element selected in the i^{th} row.

For instance, [2,3,4,1] indicates that: person-1 is assigned to job-2
person-2 is assigned to job-3
person-3 is assigned to job-4
person-4 is assigned to job-1

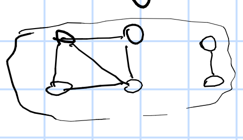


Also called marriage problem.

→ So there exist n! candidate solutions. Therefore, ES (Exhaustive Search) is impractical even for very small instances. (Hungarian Algorithm)
However Hungarian algorithm solves it in polynomial ($O(n^3)$) time.

Depth First Search / Breadth First Search

DFS & BFS systematically process all vertices and edges of a graph.
They are fundamental for detecting connectivity and cycle presence.
- To check if connected
- To check if cyclic or not cyclic



DFS

↳ Start at an arbitrary vertex and mark it as "visited".

⇒ Iteration:
- ↳ Proceed to an adjacent vertex. (Ties are resolved arbitrarily.) If there are more than 1 option, doesn't matter which to choose.
- → Continue until a dead end. (A vertex with no adjacent unvisited vertices)
- → At a dead end back 1 edge to the vertex it came from & try to continue visiting unvisited vertices from there.

The algorithm eventually halts after breaking up to the starting vertex, with the latter being a dead end. (It means done)

⊕ Accompanying a DFS traversal by constructing the DFS forest.
↳ Disconnected so apply DFS many

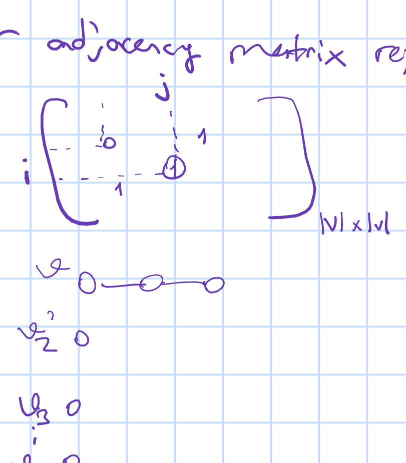
Pseudocode:

```
Algorithm DFS(G)
  count ← 0
  for each vertex v in V do
     if v is marked with 0
        dfs(v)
        count ← count+1  //Mark v with count
  for each vertex w in V adjacent to v do
        dfs(w)
```

⇒ visits recursively all unvisited vertices connected to a vertex v by a path and numbers them in the order they are encountered via a global variable "count".

• How efficient is DFS? Complexity?
• For adjacency matrix representation: $O(|V|^2)$ since there exists $|V|^2$ entries.

For adjacency list representation: $O(|V|+|E|)$



Checking Connectivity Using DFS:

When the algorithm halts check whether all vertices have been visited.

We can also use DFS to identify connected components.

Checking for a cycle presence using DFS:

If the DFS forest (notice that the DFS forest is not actually a forest) does not have back edges, the graph is acyclic.

Breadth First Search:

→ Proceeds in a concentric manner by visiting first all vertices adjacent to a starting vertex, then others etc.
→ It is convenient to use queue as a data structure.

On each iteration,
- ↳ Identify all unvisited vertices adjacent to that vertex.
- ⇒ Mark them as visited.
- ⇒ Add them to the queue and then remove the front vertex.

Pseudocode for BFS:

0 is a mark of "unvisited".

```
count = 0
for each vertex v in V do
   if v is unmarked
      bfs(v)  //visits all unvisited vertices connected to v.

count ← count + 1
Initialize a queue with v and mark v with count
while the queue is not empty do
   for each vertex w in V adjacent to the front vertex do
      if w is marked with 0
         count ← count+1, mark w with count
         add w to the queue
      remove the front vertex from the queue
```

⊕ BFS has the same efficiency as DFS.
⊕ Like DFS, BFS can be used for connectivity & acyclicity.

Decrease And Conquer

Decrease & Conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instances.

① Decrease by a constant: The size of an instance is reduced by the same constant on each iteration of the algorithm.
Typically this constant equals 1.

Ex:
① Exponentiation: Computing $a^n = a^{n-1} \cdot a$

② Decrease-by-a-constant-factor: Suggests reducing a problem instance by the same constant factor on each iteration.
In most applications, this factor equals 2.

Ex:
(2.1) Again exponentiation

$$a^n = \begin{cases} (a^{\frac{n}{2}})^2 & \text{if } n \text{ is even} \\ (a^{\frac{n-1}{2}})^2 \cdot a & \text{if } n \text{ is odd} \end{cases}$$

(2.2) Binary Search
↳ we don't care about other half (so it's not divide and conquer)
↳ it's not logarithmic

Variable Size Decrease

The size reduction pattern varies from one iteration of the algorithm to another.

Ex: Euclid algorithm for computing greatest common divisor (GCD):

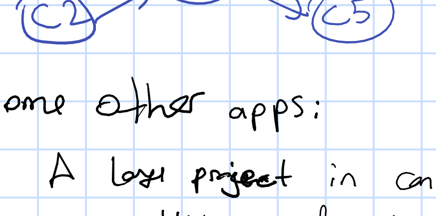
$gcd(m,n) = gcd(n, m \bmod n)$
↳ The value decreases neither by a constant nor by a constant factor.

Topological Sorting

↳ has a variety of applications involving prerequisite-related tasks. (Like courses with prerequisities)

Problem: In CS, a topological sort or topological ordering of a digraph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

Ex:



A possible topological sorting:
C1, C2, C3, C4, C5

just trying to find suitable solution.

Some other apps:

① A large project in construction, research or SW development that involves a multitude of interrelated tasks with known prerequisities.
Ex: Critical path method in SW engineering.

② Instruction scheduling in program compilation.

③ Cell evaluation ordering in spreadsheet formulas. (If there are parantheses)

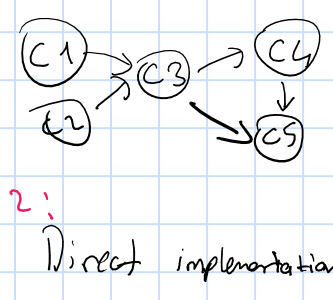
④ Resolving symbol dependencies in linkers.

Question: When does topological sorting not have a feasible solution?

Answer: When the graph have a directed cycle.
If the graph is a DAG (Directed Acyclic Graph), then it has a feasible solution.

Algorithm 1:

Algorithm 1 performs a DFS traversal and note the order in which vertices become dead ends (without outgoing edges)



C5, C4, C3, C1, C2 → Reverse order ⇒ C2, C1, C3, C4, C5
C5, C4, C3, C2, C1 → Reverse order ⇒ C1, C2, C3, C4, C5

Algorithm 2:

Direct implementation of the "decrease-and-conquer" technique:

→ Repeatedly identify in the remaining graph a source (a vertex with no incoming edges), note it down and delete it along with all edges outgoing from it.

WEEK 10

ALGORITHMS FOR GENERATING COMBINATORIAL OBJECTS

Some Types of Combinatorial Objects: Permutations, combinations, subsets of a given set.

⊛ Combinatorial Objects are studied in a branch of a discrete mathematics called combinatorics.

⊕ The # of combinatorial objects typically grows exponentially or even faster as a function of the problem size.

Our goal: Generating them, not counting them. (Read on internet: Generating permutations)

Generating Subsets

Recall the knapsack problem

Power set of n: "has cardinality 2^n"
The set of all subsets

Decrease-by-1 Idea:
All subsets of $A = \{a_1, a_2, \dots, a_n\}$ can be divided into 2 groups.

1. Sets that don't have a_n
2. " " do have a_n

→ Once we have a list of all subsets of $\{a_1, \dots, a_{n-1}\}$, we can get all subsets of $\{a_1, \dots, a_n\}$ by adding $\{a_n\}$ to each subset.

n	
Step 0	∅
Step 1	∅ $\{a_1\}$ → copy paste all → Then add
Step 2	∅ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$
Step 3	∅ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$ $\{a_3\}$ $\{a_1, a_3\}$ $\{a_1, a_2, a_3\}$

⊛ All 2^n bit strings $b_1, b_2 \dots b_n$ of length n can be generated in the same way.

Bit String	000	001	010	---	011	--	111
Subsets	∅	$\{a_3\}$	$\{a_2\}$		$\{a_1, a_3\}$		$\{a_1, a_2, a_3\}$

⊛ A challenging question is whether there exists a "minimal change algorithm" for generating bit strings so that every one of them differs from its immediate predecessor by only a single bit: Answer: Yes

Ex:
000
001
0..
⋮
..1
} → Cyclic

⇒ Such a sequence of bits is called "binary reflected Gray Code". Frank Gray @ AT&T Bell Labs reinvented it in 1940s to minimize the effect of errors in xmitting digital signals.

Algorithm Binary Reflected Gray Code (n)

```
#BRGC(n)
// Input: A positive integer n
// Output: A list of all bit strings of length n composing the gray code,

if n==1:
    make list 1 containing bit strings 0 and 1 in this order,
else:
    Generate list L1 of bit strings of size n-1 by calling BRGC(n-1)
    Copy list L1 to list L2 in reverse order
    Add 0 in front of each bit string in L1
    Add 1 in front of each bit string in L2
    Append L2 to L1 to get list L
```

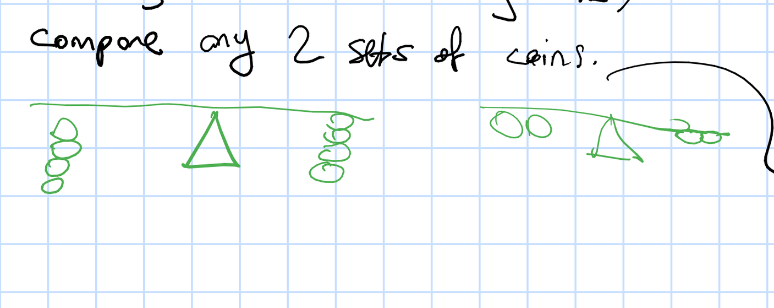
⊕ Note that BRGC is cyclic

for L1 : 0,1 —Add 0→ 00, 01
L2 : 1,0 —Add 1→ 11, 10

000 ✓
001 ✓
011 ✓
010 ✓ → 110
111
101
100

Fake Coin Problem
↳ weight of the coin

Among n identical looking coins, one is false with a balance scale; we can compare any 2 sets of coins.



↳ Continue with dividing by 2.
Divide n coins into 2 piles of $\lfloor \frac{n}{2} \rfloor$ coins each, leaving 1 extra coins aside if n is odd.

↳ Resulting recursion formula:

$$W(n) = W(\lfloor \tfrac{n}{2} \rfloor) + 1 \quad \text{for } n > 1$$
$$W(1) = 0$$

⊕ This recursion is identical to the one for the worst case number of comparisons for binary search.

⊕ The interesting point here is that the above algorithm is not the most efficient solution. It's more efficient to divide the coins into 3 piles of about $\frac{n}{3}$ coins each.

Russian Peasant Multiplication

n m ⇒ Positive integers (large numbers)

Recursive formula ⇒ If n is even ⇒ $n \cdot m = \frac{n}{2} \cdot 2m$

If n is odd ⇒ $n \cdot m = \frac{n-1}{2} \cdot 2m + m$

This way we can compute $n \cdot m$ either recursively or iteratively.

Ex:

n	m	
70	85	
35	170	
17	340	(+170) (+340)
8	680	
4	1360	
2	2720	
1	5440	

5440
170
+ 340
5950

In fact this method used in 19th century known to Egyptian mathematicians as early as 1650 BC.
It leads very fast HW implementation since doubling and halving with binary shifts.

Variable Size Decrease Algorithms

↳ The size reduction pattern varies from one iteration to another.
Ex: Euclid's algorithm for computing GCD.

Computing A Median and The Selection Problem

Selecting Problem: Finding the k^{th} smallest element in a list of n numbers.
This number is called k^{th} number statistic.
When $k = \lceil \frac{n}{2} \rceil$, this number is called the median.

Obviously, we can find it by sorting the list and then the k^{th} smallest
→ $O(n \log n)$

But wouldn't it be an overkill because we don't need the entire list but just the k^{th} smallest element? Can we do better?

⊕ Partitioning a given list around some value p:
Rearrangement such that
1. Left part contains elements less than < p
2. p: Pivot element
3. Right part contains elements greater than ≥ p

↳ We can also use other partitioning methods like quick.

Lomuto Partitioning

A subarray $A[l \dots r]$ $(0 \le l \le r \le n-1)$ composed of 3 continuous segments:

1. A segment with elements known to be < p
2. A segment with elements known to be ≥ p.
3. A segment with elements yet to be compared to p.

↳ The segments can be empty. This is always the case for the first 2 segments before the algorithm starts.

p	< p	≥ p	?
l	s	i	r (unknown segment)

Starting with i = l+1, the algorithm scans the subarray $A[l \dots r]$ left to right, maintaining this structure until a partition is achieved.

On each iteration
↳ compare the first element in the unknown segment (pointed to by i) by the pivot p.
↳ If $A[i] \ge p$: Increment i to expand the segment with elements ≥ p while shrinking the unprocessed segment.
↳ If $A[i] < p$: The segment with needs to be expanded. This is done by:
1. Incrementing the s (The index of the last element in the first segment)
2. Swapping A[i] and A[s]
3. Incrementing to point to the new first element of the shrunk unprocessed element.

Psuedocode Lomuto Partition (A[l...r])

```
p ← A[l]
s ← l
for i = l+1 to r do
    if A[i] < p
        s ← s+1
        swap (A[s], A[i])
    end if
end for
swap (A[l], A[s])
return s
```

⇒ After no unprocessed elements remain, the algorithm swaps the pivot with A[s] to achieve the partition being sought.

⊕ How can we take advantage of a list partition to find the k^{th} smallest in it?
Indices start with 0.
S = split position of the partition = Index of the array's element occupied by the pivot after partitions.

< p	p	≥ p

→ Not necessarily sorted
If k-1 simply return, if not continue

1. If $s = k-1$, pivot p is the k^{th} smallest element
2. If $s > k-1$, k^{th} smallest element is the k^{th} smallest of the left part
3. If $s < k-1$, k^{th} " " is the $(k-s)^{th}$ smallest element in the right part.

< p	p	≥ p
5 elements	s=5	3rd smallest element

k = 8
s = 5

Psuedocode for Quick Select (A[l...r], k)
Doesn't matter which partitioning you use for quickselect

```
s ← LomutoPartition (A[l...r])   # Or another partitioning algorithm
if s = k-1 return A[s]
else if s > l+k-1
    QuickSelect(A[l...s-1], k)
else
    QuickSelect(A[s+1...r], k-1)
end
```

How efficient is quick select?

Best case: Partitioning on n element array always requires n−1 key comparisons if we don't require more iterations, this the best case.

$$C_{best}(n) = n-1 \in \Theta(n)$$

Worst case: An extremely unbalanced partition (with 1 part empty and the other part (n−1) elements)
↳ This can happen on each of the (n−1) iterations
⇓
$$C_{worst}(n) = (n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

⊕ A careful mathematical analysis shows that the average case is linear.

⊕ A more sophisticated way of choosing a pivot guarantees linear time even in the worst case (proved in 1973)

⊕ Notice that the partitioning-based algorithm identifies the smallest and the k smallest and n−k largest elements, not just the value of the k^{th} smallest element.

WEEK 11

DIVIDE & CONQUER

1. Divide the problem into several subproblems of the same type, ideally about equal size.
2. Solve the subproblems (typically recursively)
3. Combine the solutions to the subproblems to a solution to the original problem.

→ Spend only linear time for the initial division and final recombining.

⊕ In many settings where divide and conquer is applied, the natural brute force algorithm may already be poly-time. In this case, D & C strategy may (↳ Not always) serve to reduce running time to a lower polynomial.

Ex: (for not-always-work)
Computing sum of n numbers
→ Compute sum of the first $\lfloor \frac{n}{2} \rfloor$ numbers.
→ Compute " " " " $\lceil \frac{n}{2} \rceil$ numbers.
→ Add their values to get the sum in question.

Ex: The natural BF approach for closest pair problem has complexity $\Theta(n^2)$
D & C approach (we'll see) yields $O(n \log n)$

⊕ Typical recurrence for D & C:

$T(n) = a \cdot T(\frac{n}{b}) + f(n)$

↳ $f(n)$: A function that accounts for dividing the instance of size n into instance of size $\frac{n}{b}$ & combining their solutions.

⊕ If $a = 1$, D & C covers **decrease by a constant-factor** algorithms.

Merge Sort

→ Divide the array into 2 halves, sort each one recursively and merge them to a single sorted one.

Algorithm Merge Sort (A[0.....n-1])

```
if n > 1
   copy A[0,.....,⌊n/2⌋-1] to B[0,....⌊n/2⌋-1]
   copy A[⌊n/2⌋ .....n-1] to C[0,...,⌈n/2⌉-1]
   Merge Sort (B[0....⌊n/2⌋-1])
   Merge Sort (C[0....⌈n/2⌉-1])
   Merge (B,C,A)
```

⊕ The merging of 2 sorted arrays can be done as follows:
⊕ 2 pointers are initialized to point to the first elements of the arrays.
⊕ Compare the elements pointed to, copy the smaller one to the new array, & increment the index of the smaller element.

Algorithm Merge (B[0...p-1], C[0....q-1], A[0....p+q-1])

```
i ← 0, j ← 0, k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]
        i ← i+1
    else
        A[k] ← C[j]
        j ← j+1
    end
    k ← k+1
if i = p    # just put the rest
    Copy C[j...q-1] to A[k...p+q-1]
else
    Copy B[i...p-1] to A[k...p+q-1]
```

Ex: 6, 5, 7, 1, 8, 2, 10, 4

6,5   7,1   8,2   10,4

6 5   7 1   8 2   10 4

Compare

[5 6] [1 7] [2 8] [4 10]

[1 5 6 7]   [2 4 8 10]

[1, 2, 4, 5, 6, 7, 8, 10]

$C(n) = 2 \cdot C(\frac{n}{2}) + C_{merge}(n)$

At each step, exactly one comparison is needed.
In the worst case, neither of the 2 arrays become empty before the other one contains just 1 element.

Hence, $C_{merge}(n) = n-1$ and we have $C_{worst}(n) = 2\, C_{worst}(\frac{n}{2}) + n - 1$

⇓

$C_{average} = O(n \log n)$

⊕ What if n is odd;

$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + cn$ (↳ $n-1$)

⇓

Asymptotic bounds don't change.

Multiway Mergesort: Divide a list to be sorted in more than 2 parts, sort each one recursively, and then merge them together.

Ex: $T_{3\text{-way mergesort}} = 3T(\frac{n}{3}) + n - 1$; $a = 3$, $b = 3$

↳ $f(n) \in \Theta(n)$, $d = 1$

$a = 3 = 3^1 \Rightarrow T(n) \in \Theta(n \log n)$

⊕ A noteworthy advantage of mergesort over quicksort is its stability.

⊕ The principal shortcoming of mergesort is the linear amount of extra storage the algorithm requires.

QUICKSORT

Unlike mergesort, which divides its input elements according to their position, quicksort divides them according to their value.

Partition ⇒ A partition is an arrangement of the array's elements so that all elements to the left of some element A[s] are less than A[s] and all elements to the right of some element A[s] are greater than A[s].

→ In mergesort, the division is obvious and the entire work happens in combining their solutions.

→ In quicksort, the entire work happens in the division state with no work required to combine the solutions to the algorithm.

Algorithm Quicksort

```
Algorithm Quicksort (A[l...r])
if l < r
    s ← Partition (A[l...r])    // s is split position
    Quicksort (A[l...s-1])
    Quicksort (A[s+1...r])
```

→ As a partitioning algorithm, we can certainly use Lomuto Partition or a more sophisticated method by Hoare as follows;

→ We start by selecting a pivot.
For now, do the simplest selection. Choose $p = A[l]$

⊕ Unlike the Lomuto algorithm, now scan the subarray from both ends, comparing the subarray's elements with the pivot.

i: Denotes the left-to-right scan and starts with the second element. This scan stops upon encountering the first element ≥ pivot.

j: Denotes the right-to-left scan and starts with the last element. This scan stops upon encountering the first element ≤ pivot.

⊕ After both scans stop, 3 situations may arise;

1. If i and j have not crossed (ie $i < j$), we simply exchange A[i] and A[j] and resume the scans by incrementing i and decrementing j.
2. If they have crossed over (ie $i > j$), we will have partitioned the subarray after exchanging the pivot with A[j]:
   | p | All are ≤ p | ≤ p | ≥ p | All are ≥ p |
3. If i and j stop while pointing to the same element (ie $i = j$), the value they are pointing to must be equal to p. Thus we already have the subarray partitioned with the split position $s = i = j$.
   | p | All are ≤ p | = p | All are ≥ p |

Algorithm Hoare Partition

```
Algorithm HoarePartition (A[l...r])
    p ← A[l]  // Pivot
    i ← l
    j ← r+1
    repeat
        repeat i ← i+1 until A[i] ≥ p
        repeat j ← j-1 until A[j] ≤ p
        swap (A[i], A[j])
    until i ≥ j
        swap (A[i], A[j])   // Undo the last swap when i ≥ j
        Swap (A[l], A[j])
        return j
```

Efficiency: The number of key comparisons made before a partition is achieved $= \begin{cases} n+1, & \text{if the scanning indices cross over} \\ n, & \text{if they coincide} \end{cases}$

→ If all splits happen in the middle of the corresponding subarrays, we have the best case.

$C_{best}(n) = 2 \cdot C_{best}(\frac{n}{2}) + n$ for $n > 1$, $C_{best}(1) = 0$

⇓

According to the master theorem, $C_{best}(n) \in \Theta(n \log n)$
Solving exactly for $n = 2^k$ yields: $C_{best}(n) \in \Theta(n \log n)$

Worst case:
All splits are skewed to the extreme;
1 array is empty
The other subarray has size just 1 less than the size of subarray being partitioned.
⇓
This situation occurs for inputs th

Efficiency:
The number of key comparisons made before a partition is achieved ⇒ $n+1$, if the scanning indices cross over; n, if they coincide

Master Theorem
→ $C_{best}(n) = 2\, C_{best}(\frac{n}{2}) + n$ for $n > 1$, $C_{best}(1) = 0$

⇓

$C_{best}(n) \in \Theta(n \log n)$

Worst Case:
All splits are skewed to the extreme.
for already skewed inputs

| A[0] | A[1] | ---- | A[n-1] |

← j → i

(n+1) comparisons in iteration 1
(n) " in " 2
⋮
3 ---

$\frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2)$

Average Case:
A partition can happen in any position s ($0 \le s \le n-1$) after n+1 comparisons are made to achieve the partition.

(0 ... n-1: s elements | n-1-s elements)

⊗ Assuming that the partition split can happen in each position s with the same prob $\frac{1}{n}$, we get the following recurrence;

$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} \left[(n+1) + C_{avg}(s) + C_{avg}(n-1-s) \right]$ for $n > 1$

$C_{avg}(0) = 0$
$C_{avg}(1) = 0$

↳ Full history recurrence
↳ $C_{avg}(n) \in \Theta(n \log n)$

⊗ Better pivot selection method
⊕ Switching to insertion sort on very small subarrays. (5-15 elements)
⊕ Quicksort is not stable.

Counting Inversions

In worst case, it's reverse sorted.

2 indices $i < j$ form an inversion if $a_i > a_j$ ("out of order")

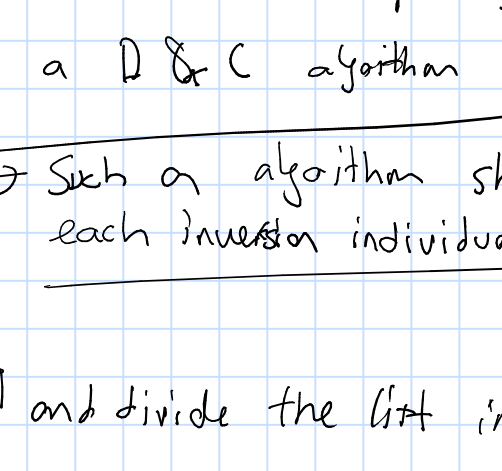
$\frac{n(n-1)}{2} = \binom{n}{2}$

Ex: Colaboritive Filtering:
They try to match your preferences (for books, movies, etc) with those of other people out on the internet.
⇒ Identifying people with "similar" interests.
Another app: Metasearch
↳ execute the same query on many different search engines & then synthesize the results by comparing the ranking.

A core issue;
The problem of comparing 2 rankings. What's a good way to measure how similar 2 people's ranking case?

⊕ Given a sequence of n distinct numbers $a_1, \ldots, a_n$, determine the number of inversions in the sequence.

Geometric way;



⇒ Each crossing pair of line segments corresponds to 1 pair that is in the opposite order in the 2 lists.

In ascending order: 0 inversions
In decreasing order: $\frac{n(n-1)}{2}$ inversions

Designing & Analysing The Algorithm

Simplest algorithm (ES): Check each pair ⇒ $O(n^2)$ time
Now we'll show a D & C algorithm to count them in $O(\log n)$ time.
→ Such an algorithm should behave without looking at each inversion individually.

→ Set $m = \lceil \frac{n}{2} \rceil$ and divide the list into the 2 pieces $a_1, \ldots, a_m$ and $a_{m+1}, \ldots, a_n$.
→ Then count the # of inversions (a_i, a_j) where the 2 numbers belong to different halves.
→ We must do this part $O(n)$ time.

⊕ These inversions have a nice form: They are precisely the pairs (a_i, a_j) where a_i is in the first half and a_j is in the second half; and $a_i > a_j$.

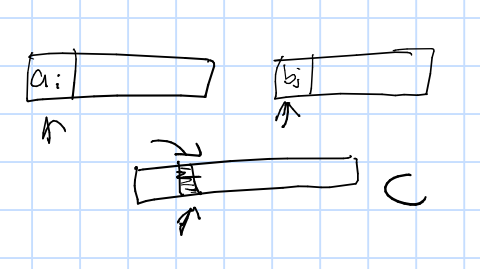
⊕ We will make the algorithm recursively sort the numbers in the 2 halves as well.

→ After sorting, we have 2 sorted lists, (A and B); containing first and second halves, respectively. We want to produce a single sorted list C from their union, while also counting the number of pairs (a,b) with $a \in A$, $b \in B$ and $a > b$.

WEEK (12) 14

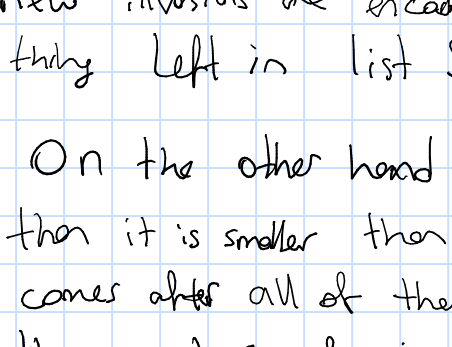
→ In each step, have a current pointer into each list. Suppose that those pointers are currently at elements.

→ In each step, compare the elements that a_i and b_j being pointed to in each list, remove the smaller one from the list and append it to the end of list C.



→ Every time the element a_i is appended to C, no new inversions are encountered since a_i is smaller than everything left in list B.

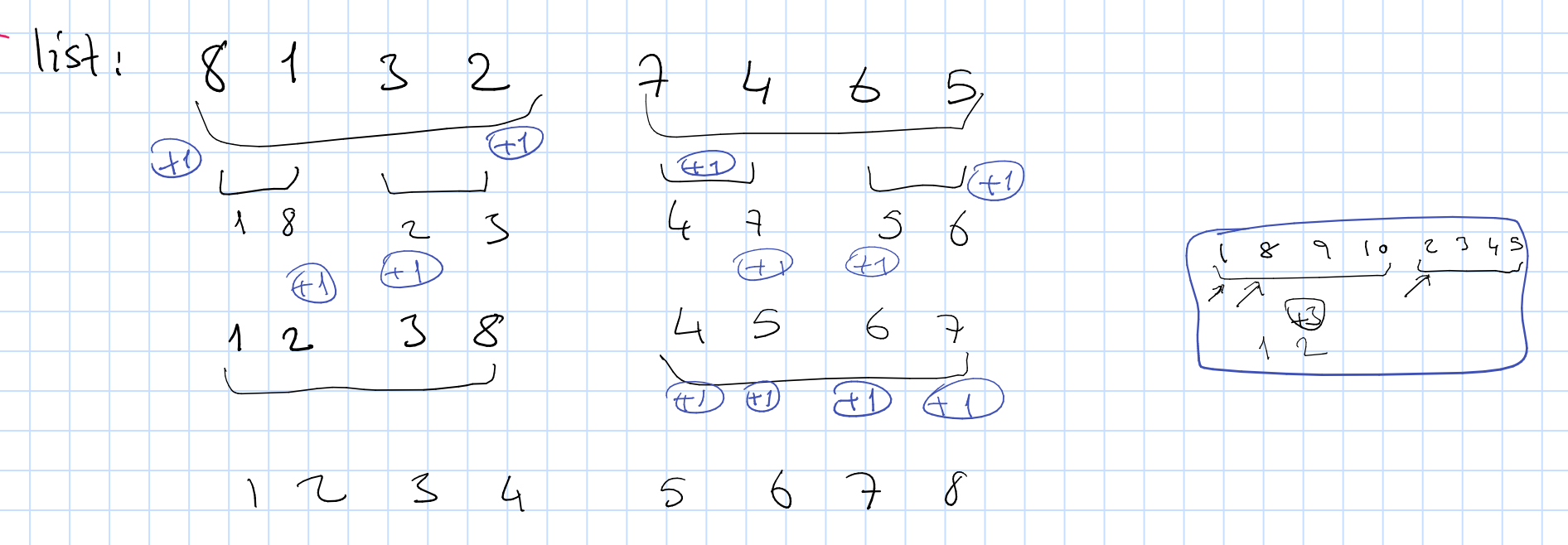
→ On the other hand, if b_j is appended to list C, then it is smaller than all the remaining items in A and comes after all of them. So we increase our count of the number of inversions by the number of remaining elements in A.



we don't have to compare with others.

This way in constant time, we have accounted for a potentially huge number of inversions.

Ex:



Pseudocode:

Merge_and_Count (A, B)

→ Maintain a current pointer into each list initialized to point to the front elements.

→ Maintain a variable count for the number of inversions, initialized to 0.

→ While both lists are nonempty:

- → Let a_i and b_j be the elements pointed to by the current pointer.
- → Append the smaller of these two to the output list.
- → If b_j is the smaller element then; increment count by the number of elements remaining in A
  end if
- → Advance the current pointer in the list from which the smaller element was selected.
- → Once the list is empty, append the remainder of the other list to the output.
- → Return Count and the merged list.

Running time of Merge_and_Count:

- → Each iteration of the while loop takes constant time.
- → In each iteration, we add some element to the output that will never be seen again.

Thus, the number of iterations can be at most the sum of the initial lengths of A & and B, so the total running time is $O(n)$.

Pseudocode for Sort_and_Count (L)

If the list has 1 element, then ∄ no inversions
else:
Divide the list into two halves:
A contains the first $\lceil \frac{n}{2} \rceil$ elements
B contains the remaining $\lfloor \frac{n}{2} \rfloor$ elements

$(r_A, A) = sort\_and\_count(A)$
$(r_B, B) = sort\_and\_count(B)$
$(r, L) = Merge\_and\_count(A, B)$
Return $r \Leftarrow r_A + r_B + r$ and the sorted list L

⇒ Since Merge and Count takes $O(n)$ time, the entire running time satisfies the recurrence:

$$T(n) \leq 2T(\tfrac{n}{2}) + c_n$$
$$\Downarrow$$
$$T(n) \in O(n \log n)$$

FINDING THE CLOSEST PAIR OF POINTS

The set of points ⇒ $P = \{p_1, \ldots, p_n\}$, where p_i has coordinates $\{x_i, y_i\}$

$d(p_i, p_j)$: The euclidian distance between two points.

Goal: To find a pair of points p_i, p_j that minimizes $d(p_i, p_j)$

→ Assume that no 2 points have the same x or y coordinate.

→ Divide the area into 2 halves

Find the closest pair in the left half.
Find the closest pair in the right half.
} Then use this information to get the overall solution in linear time. (Combining step)

Combining Step:

The distances that have been considered by either of the recursive calls are precisely those points that occur between a point in the left half and a point in the right half.

⇓

There exists $\Omega(n^2)$ such distances and we need to find the smallest one in $O(n)$ time.

On a set $P' \subseteq P$, every recursive call begins with 2 lists:

→ A list P'_x in which all points in P' have been sorted by increasing x coordinate.

→ A list P'_y in which all points in P' have been sorted by increasing y-coordinate.

$Q \triangleq$ The set of points in the first $\lceil \frac{n}{2} \rceil$ positions of the list P_x. (the "left half")

$R \triangleq$ The set of points in the final $\lfloor \frac{n}{2} \rfloor$ " " " " P_x. (the "right half")

→ By a single pass through each of P_x and P_y (in $O(n)$ time), we can create the following 4 lists:

1. $Q_x \triangleq$ (Definition) Points in Q sorted by increasing x coordinate.
2. $Q_y \triangleq$ Points in Q sorted by increasing y cordinate.
3. $R_x \triangleq$ (Definition) Points in R sorted by increasing x coordinate.
4. $R_y \triangleq$ Points in R sorted by increasing y cordinate.

For each entry (point), we record its position in both lists it belongs to.

Combining the Solution:

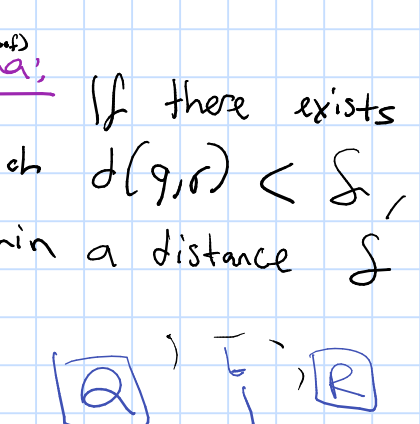
q_0^* and q_1^*: Closest pair of points in Q.
r_0^* and r_1^*: " " " " " R.

Let $\delta = \min\{d(q_0^*, q_1^*), d(r_0^*, r_1^*)\}$

→ Are there points $q \in Q$ and $r \in R$ for which $d(q, r) < \delta$

→ $x^* \triangleq$ x coordinate of the rightmost point in Q.

Lemma: (Part of proof) If there exists $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$, then each of q and r lies within a distance δ of L.



Proof: Suppose that such q and r exists. We write $q = (q_x, q_y)$ and $r = (r_x, r_y)$

By definition of x^*, we know that $q_x \leq x^* \leq r_x$.

Then we have;

$$x^* - q_x \leq r_x - q_x \leq d(q,r) < \delta \text{ (delta)}$$
$$r_x - x^* \leq r_x - q_x \leq d(q,r) < \delta$$

⇒ So each of q and r has an x coordinate within δ of x^* and lies within distance δ of the line L.

⇒ So if we want to find a close q and r, we can restrict our search to the narrow band consisting only of points in P within δ of L.

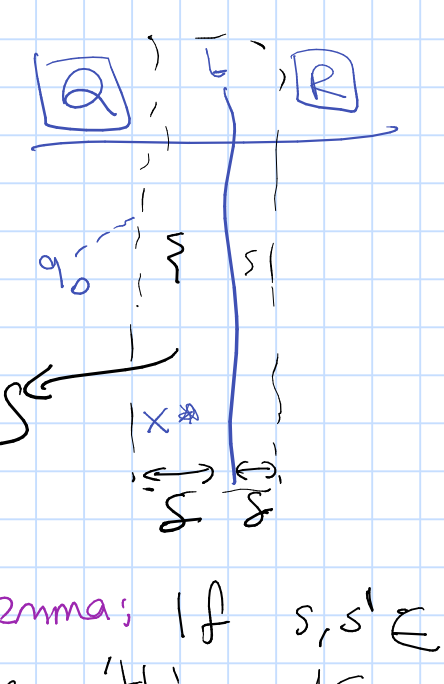
Let $S \subseteq P$ denote this set (set of points in P within δ of L.)

S_y: Set of points in S sorted by increasing y coordinate.

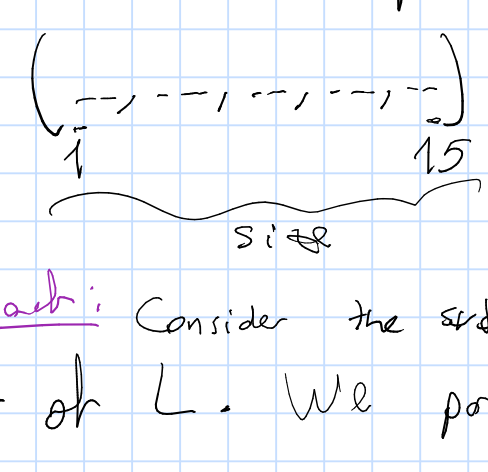
⇓

By a single pass through the list P_y, we can construct S_y in $O(n)$ time.

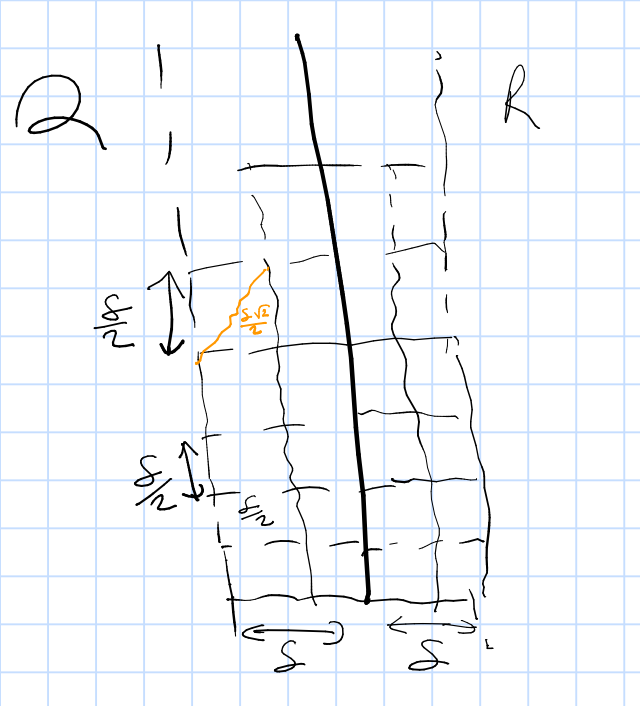
— A restatement of the previous lemma: There exist $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$ iff there exist $s, s' \in S$ for which $d(s,s') < \delta$.



Lemma: If $s, s' \in S$ have the property that $d(s,s') < \delta$, then s and s' are within 15 positions of each other in the sorted list S_y.



Proof: Consider the subset Z of the plane consisting of all points within distance δ of L. We partition into boxes as in here.

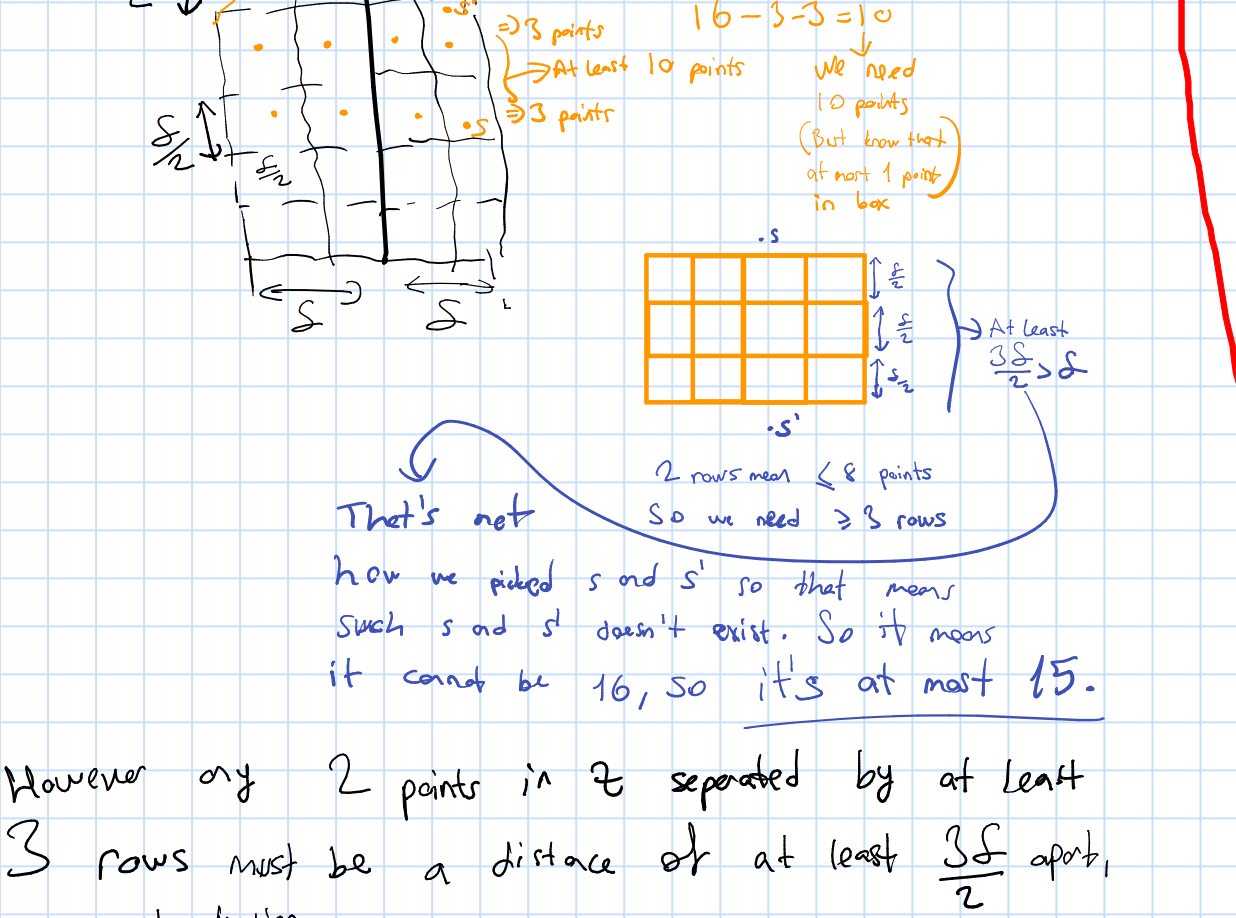


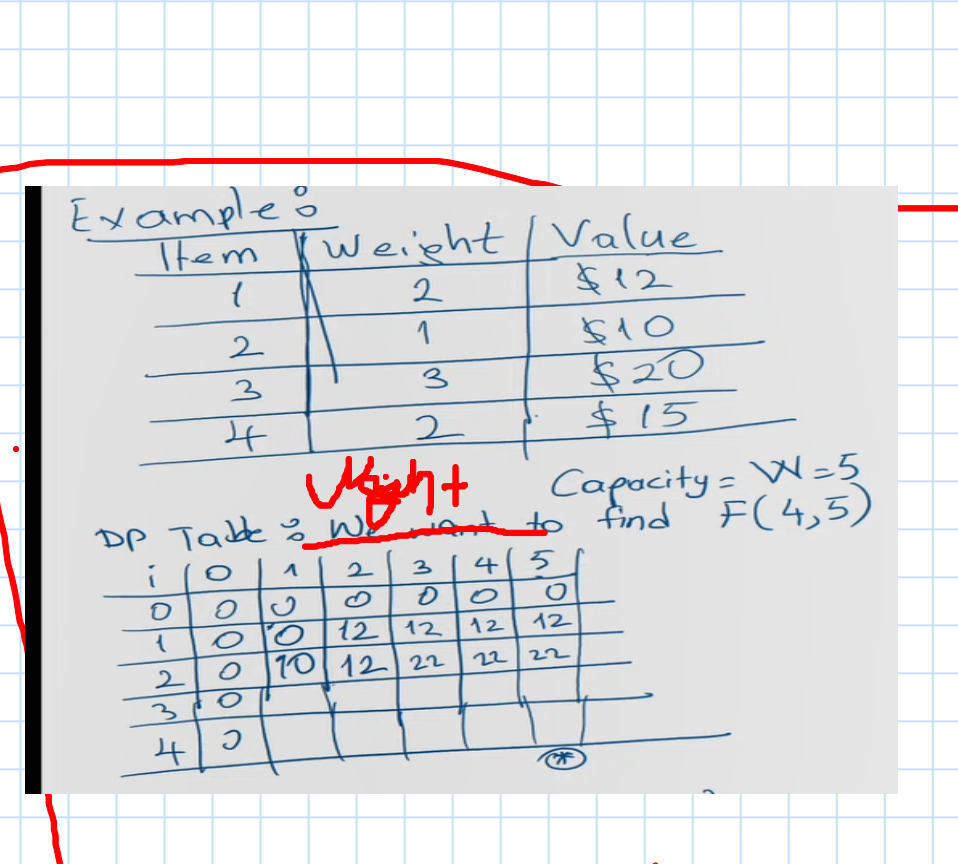
We first prove that, each box contains at most 1 point of S.

Suppose that 2 points of S lie in the same box. Since all points in this box lie on the same side of the line L, these 2 points either both belong to Q or both belong to R.

However, any 2 points in the same box are within distance $\frac{\delta\sqrt{2}}{2} < \delta$, which contradicts with our definition of δ as the minimum distance between any pair of points in Q or in R. Thus, each box contains at most 1 point of S.

Now, suppose that $s, s' \in S$ have the property that $d(s, s') < \delta$ and that they are at least 16 positions apart in S_y. Assume without loss of generality (wlog) that s has the smaller y coordinate. (Let's say s is smaller. If we say s' is smaller doesn't change anything.) Then, since there can be at most 1 point per box, there exist at least 3 rows of Z lying between s and s'.




Example:

Item	Weight	Value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

Capacity = W = 5
DP Table: Weight to find F(4,5)

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0					
4	0					

However any 2 points in Z separated by at least 3 rows must be a distance of at least $\frac{3\delta}{2}$ apart, a contradiction.

⊕ This value 15 can be reduced. However for our purposes the important thing is that it is an absolute constant.

⊕ We make 1 pass through S_y and for each $s \in S_y$, we compute the distance to each of the next 15 points in S_y. This way, we compute the distance of each pair of points S that are at distance less than δ from each other.