Algorithmic Toolbox

Introduction

- Fibonacci number F_n
  - recursive is very slow as compute $F_{n-2}, F_{n-3}, \cdots$ multiple times
  - more efficient to memorize $F_1, F_2, \cdots$ and compute $F_{n-1} + F_{n-2}$ to get F_n

- Greatest Common Divisor (GCD)
  → the largest integer d that divides a and b

Key Lemma

Let a' be remainder when a is divided by b then

$$gcd(a, b) = gcd(a', b)$$

Why? $a = a' + bq$

if d divides a and b, d must also divide a' and b

↓

Euclidean algorithm

```
def EuclidGCD(a, b):
    if b=0:
        return a
    a' = a%b        # a mod b
    return EuclidGCD(b, a')
```

NO.

DATE

Runtime: - each step reduces the size of number by a factor of 2

→ log (ab) steps

> * Finding the correct algorithm requires knowing something interesting about the problem.

Debugging

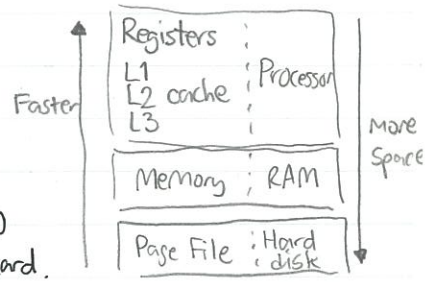
* practice good testing techniques
  1. Small manual tests
  2. test for each possible type of answer (e.g. max, min, impossible)
  3. test for time/memory limit by generating largest possible input size
  4. test for corner cases
     - smallest possible n
     - * equal numbers / letters / objects
     - largest input allowed → careful of integer overflow!
     - degenerate cases e.g. empty sets, 3 points on a line, trees w/o branches disconnected graphs, full graph, etc.
  5. stress testing
     - → solution you want to test
     - → diffirent (e.g. very slow but surely correct) solution
     - → test generator
     - → infinite loop to check consistency between two solutions

DATE

Computing Runtime

- Obviously, # of lines is not the most representative measure of runtime
- But actual runtime depends on:
  - Speed of computer
  - System architecture
  - the compiler being used
  - details of memory hierarchy

→ messy! Computing runtime is hard.



Asymptotic Notation

All these issues multiply runtimes by constant

→ measure runtime in a way that ignore constant multiples

↓

Consider asymptotic runtime (How does it scale with input size?

$$\log n \prec \sqrt{n} \prec n \prec n\log n \prec n^2 \prec 2^n$$

Big-O Notation

"is equal to" よりも "is" (つまり ∈ の方に近い)

$f(n) = O(g(n))$ or $f \preceq g$ if $\exists N, c \in \mathbb{R}$ s.t.

$\forall n \geq N,\ f(n) \leq c \cdot g(n)$

(f is bounded above by some constant multiple of g)

NO.

DATE

(3.1) $3n^2 + 5n + 2 = O(n^2)$

Since for $n \geq 1$, $3n^2 + 5n + 2 \leq 3n^2 + 5n^2 + 2n^2 = 10n^2$

Advantages

- identify growth rate
- clean up notation
- ignore complicated details

* Base of log doesn't matter as it differs by constant.

Disadvantage

- ignore constant factors
- only asymptotic → doesn't say anything about practical input size

Some rules

- $n^a \prec n^b$ for $0 < a < b$
- $n^a \prec b^n$ for $a > 0$, $b > 1$
- $(\log n)^a \prec n^b$ for $a, b > 0$

Other notations

- $f(n) = \Omega(g(n))$ or $f \succeq g$ if $\exists c$, $f(n) \geq c \cdot g(n)$
  → f grows no slower than g
- $f(n) = \Theta(g(n))$ or $f \asymp g$ if $f = O(g)$ and $f = \Omega(g)$
  → f grows at the same rate as g
- $f(n) = o(g(n))$ or $f \prec g$ if $\lim_{n \to \infty} f(n)/g(n) = 0$
  → f grows slower than g

DATE

Example: Determining Fibonacci F[N]

Operation Fibonacci(N):	Runtime
F = range(N + 1) # create array	O(n)
F[0] = 0	O(1)
F[1] = 1	O(1)
for i in range(2, N):	Loop O(n) times
F[i] = F[i-1] + F[i-2]	O(n)
return F[n]	O(1)

constant time

Normally addition is O(1) but for large numbers (recall that Fibonacci gets extremely large), the amount of work $\propto$ # of digits and # of digits $\propto N \Rightarrow O(n)$

$O(n) + O(1) + O(1) + O(n) \cdot O(n) + O(1) = O(n^2)$

豆知識

- Fibonacci $F_n \bmod m$ is always periodic, starting with 01 ..... [Pisano period].

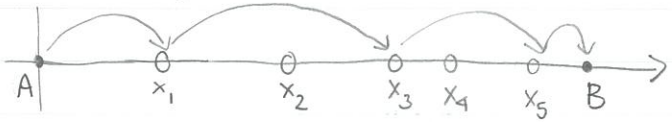
Greedy Algorithm (欲張り法)

- Make some greedy choice
- Proof that it's a safe move
- Reduce to a subproblem (similar prob with smaller size)
- Solve the subproblem (Iterate)

A greedy choice is a safe move if there's an optimal solution consistent with this first move

* Not all first moves are safe.
* Often greedy choice isn't safe.

Example 1: Car Fueling



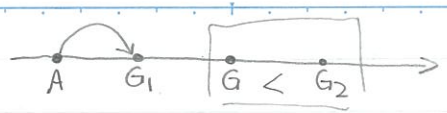
- Traveling from A to B
- Max distance w/ full tank = L
- Gas stations at $x_1 \leq x_2 \leq \dots \leq x_n$

Q: The minimum number of refills = ?

Greedy choice => Refill at the farthest reachable gas stn (G)
Subproblem => Make G the new A
this is a safe move. Proof: Let route R be the optimal route.

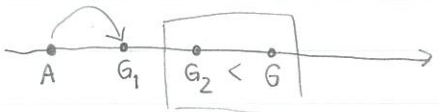
- G_1 is the first refill in R
- G_2 is the next stop in R
- G is the farthest reachable station

1st scenario:



→ Refuel at G instead of G_1

2nd scenario:



→ then avoid G_1 and refill instead at G_2
→ G_1 is not part of the optimal route R
→ contradictory Q.E.D.

Implementation: $A = x_0 \leq x_1 \leq \dots \leq x_n \leq x_{n+1} = B$

```
def MinRefills(x, n, L):
    numRefills = 0          # not counting initial position
    curRefills = 0
    while curRefills ≤ n:                    # stop if reach B (n+1)
        lastRefills = curRefills
        while (curRefills ≤ n and x[curRefills + 1] - x[lastRefills]
               ≤ L):                         # is the next station reachable?
            curRefills += 1    # move till farthest reachable
        if lastRefills == curRefills:
            return -1          # impossible
        if curRefills ≤ n:
            numRefills += 1
    return numRefills
```

Runtime: O(n) not $O(n^2)$ even though there're two loops. (everything else is constant time)

Why? CurRefills increases one by one until n+1. → linear to n
numRefills " " mostly n.

Example 2: Children Grouping

Problem: organize children into group such that the age difference of any two children in a group ≤ 1. What is min(# of groups)?

⇒ Input: $x_1, x_2, \cdots, x_n \in \mathbb{R}$

Output: min(# of segments with length = 1 s.t. cover all points x_i)

Naive algorithm: check all possible permutations ... runtime = $\Omega(2^n)$

(proof: if there're 2 partitions, # of possible ways to group = 2^n) at least

Greedy algorithm:

Safe move: cover leftmost point w/ a unit segment w/ left end at this point

Assume $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$

```
def PointsCoverSorted(X):
    i, r = 0, 0
    while i < n:
        r = X[i] + 1
        while i < n and x[i] ≤ r:
            i += 1
    return r
```

Running time is O(n) as i changes from 0 to n, each i belongs to at most one segment.

- but must sort x_i → $O(n \log n)$

→ Total running time = $O(n \log n) + O(n) = O(n \log n)$

Example 3: Fractional Knapsack

Input: Weights $w_1, w_2, \cdots, w_n$, Total Capacity W

Values $v_1, v_2, \cdots, v_n$

Output: max value that fits in capacity W (items can be fractions) → divisible

safe move → pack as much as possible items with high value per weight.

Two ways to implement:

```
def Knapsack1(W, w, v):
    A = [0]*n
    V = 0
    for i in range(n):          # main loop is executed n times
        if W == 0:
            return (V, A)
        i = item with w_i > 0, max v_i/w_i
        a = min(w_i, W)
        V, w_i, A[i], W = (V + a*v_i/w_i), (w_i - a), (A[i] + a), (W - a)
    return (V, A)
```

Selecting best item is O(n)

⇓

Running time is $O(n^2)$ ... not optimal

```
def Knapsack2(W, w, v)          # sort first!
    A = [0]*n
    V = 0
    Sort(w, v) # sorting by v_i/w_i desc     # Sorting is O(n log n)
    for i in range(n):
        if W == 0:
            return (V, A)
        a = min(w_i, W)
        V, w_i, A[i], W = (V + a*v_i/w_i), (w_i - a), (A[i] + a), (W - a)
    return (V, A)
```

each iteration is O(1) executed n times → O(n)

⇓

Runtime = $O(n \log n) + O(n)$

$= O(n \log n)$!

... Optimal!

* Greedy algorithm can be faster after sorting!

Divide-and-Conquer (分割統治法)

Main idea: 1. Break problem into non-overlapping subproblems of the same type
2. Solve subproblems ⋯→ base case
3. Combine results

- Linear Search → trivial case of divide-and-conquer

Problem: input ⇒ Array A with n elements, key k
output ⇒ an (not "the" as there can be duplicates) index i where A[i] = k, if there's no such i then NOT_FOUND.

```
def LinearSearch(A, low, high, key):
    if high < low:
        return -1        # NOT_FOUND
    if A[low] == key:
        return low
    return LinearSearch(A, low+1, high, key)
```

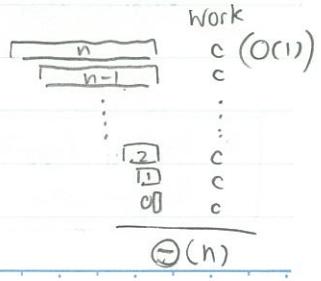
base case

New problem is smaller by 1

best case: first item is a match.

Recurrence relation for worst case scenario (not found):

$T(n) = T(n-1) + c$ ← constant time checking high<low, checking if key matches
⋮
$T(0) = c$

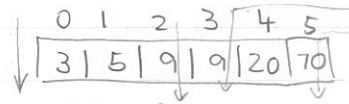


Runtime = $\Theta(n)$

- Binary Search
  - Searching sorted data

Problem: input: Sorted Array A, key k ∀i A[i] ≤ A[i+1]
output: index i where A[i] = k otherwise the greatest index i where A[i] < k. If k < A[0] then -1.

e.g.



search(13) = 3
search(99) = 5
search(1) = -1
search(9) = 2 or 3

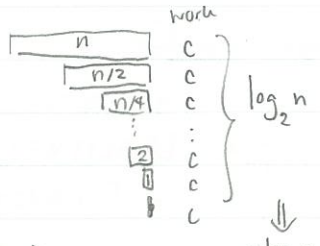
<Recursive Version>

```
def BinarySearch(A, low, high, key):
    if high < low:
        return low - 1
    mid = low + (high-low) // 2
    if key == A[mid]:
        return mid
    elif key < A[mid]:
        return BinarySearch(A, low, mid-1, key)
    else:
        return BinarySearch(A, mid+1, high, key)
```

$\lfloor \frac{high-low}{2} \rfloor$

Runtime:

$T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + c$
⋮
$T(0) = c$



Runtime = $\Theta(\log n)$

<Iterative version>

```
def BinarySearch2(A, low, high, key):
    while low <= high:
        mid = low + (high-low) // 2
        if key == A[mid]:
            return mid
        elif key < A[mid]:
            high = mid - 1
        else:
            low = mid + 1
    return low - 1
```

Polynomial Multiplication

Uses: error-correcting codes
- large-integer multiplication
- convolution in signal processing

input: $a = (a_{n-1}, a_{n-2}, \cdots, a_1, a_0)$ representing $a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$

$b = (b_{n-1}, b_{n-2}, \cdots, b_1, b_0)$ $\quad$ $b_{n-1}x^{n-1} + \cdots + b_1 x + a_0$

output: $c = (c_{2n-2}, c_{2n-1}, \cdots, c_1, c_0)$ $\quad$ $c_{2n-2}x^{2n-2} + \cdots + c_1 x + c_0$

$\cdots a_1b_0 + a_0b_1 \quad a_0b_0$

e.g. $a = (3,2,5)$, $b = (5,1,2) \rightarrow c = (15, 13, 33, 9, 10)$ (3×2 + 2×1 + 5×5, 5×2)

- Naive algorithm

```
def MultPoly(A,B,n):
    pair = [[A[i]*B[j] for i in range(n)] for j in range(n)]
    product = [0] * (2n-1)
    for i in range(n):
        for j in range(n):
            product[i+j] += pair[i][j]
    return product
```

n^2 multiplications, n^2 additions → $O(n^2)$

- Naive divide and conquer algorithm

Assume n Mod 2 = 0, if not, just add a new term $0 \cdot x^n$

$A(x) = D_1(x)x^{\frac{n}{2}} + D_0(x)$ where $D_1(x) = (a_{n-1}, \cdots, a_{n/2})$

$D_0(x) = (a_{\frac{n}{2}-1}, \cdots, a_0)$

Likewise, $B(x) = E_1(x)x^{\frac{n}{2}} + E_0(x)$ where $E_1 = (b_{n-1}, \cdots, b_{n/2})$

$E_0 = (b_{\frac{n}{2}-1}, \cdots, b_0)$

Then, $A(x)B(x) = (D_1E_1)x^n + (D_1E_0 + D_0E_1)x^{n/2} + D_0E_0$

e.g. $A(x) = 4x^3 + 3x^2 + 2x + 1 \rightarrow D_1(x) = 4x+3$ $\quad D_0(x) = 2x+1$

$B(x) = x^3 + 2x^2 + 3x + 4 \rightarrow E_1(x) = x+2$ $\quad E_0(x) = 3x+4$

$AB = (4x^2+11x+6)x^4 + (12x^2+25x+12 + 2x^2+5x+2)x^2 + (6x^2+11x+4)$

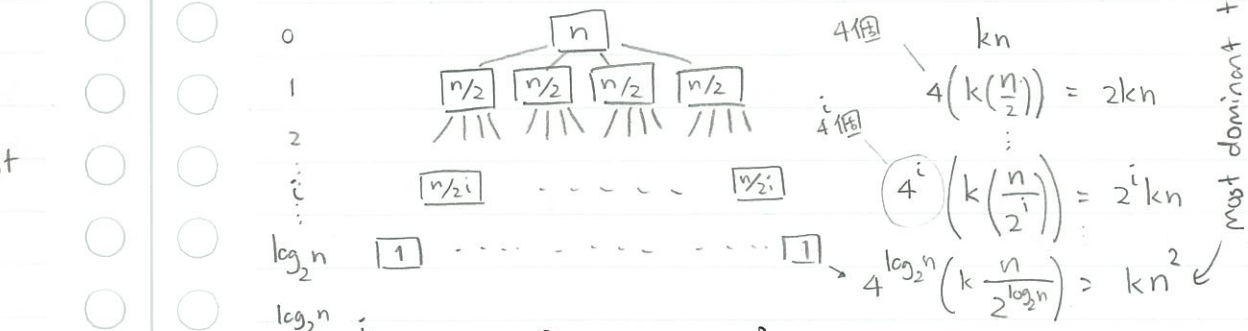
D_1E_1 $\quad D_1E_0$ $\quad D_0E_1$ $\quad D_0E_0$

```
def Mult2(A,B,n):   # Assume n is power of 2.
    R = array[0, ..., 2n-2]
    if n == 1:
        R[0] = A[0] * B[0]
    Return R
    R[0,...,n-2] = Mult2(A[0 ... n/2], B[0 ... n/2], n/2)
    R[n,...,2n-2] = Mult2(A[n/2 ... n], B[n/2 ... n], n/2)
    R[n/2,...,n+n/2-2] += Mult2(A[0,...,n/2], B[n/2 ... n], n/2) +
                          Mult2(A[n/2,...,n], B[0 ... n/2], n/2)
    return R
```

(偽コード)

1 mult → 4 mult

$T(n) = 4T\left(\frac{n}{2}\right) + kn$ (additions, calculate mid point, etc.)



$\sum_{i=0}^{\log_2 n} 2^i kn \rightarrow$ Runtime is $\Theta(n^2)$, no improvement over naive algo!

NO.

DATE

Karatsuba divide-and-conquer algorithm

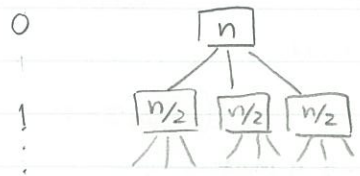
→ faster than naive divide-and-conquer!

Intuition: $AB = (D_1E_1)x^n + (D_1E_0 + D_0E_1)x^{n/2} + D_0E_0$ 4 mult

$$(D_1 + D_0)(E_1 + E_0) - D_1E_1 - D_0E_0$$

3 mult!

$$= (D_1E_1)x^n + ((D_1 + D_0)(E_1 + E_0) - D_1E_1 - D_0E_0)x^{n/2} + D_0E_0$$



k → the work at each step increases as there's additional process but still linear time (additions)

$(3^1)k\frac{n}{2}$

$(3^{\log_2 n})k$ ↓ but asymptotically faster!!

$\log_2 n$ [1] . . . [1]

$$\sum_{i=0}^{\log_2 n} 3^i k \frac{n}{2^i} = \Theta(n^{\log_2 3}) \rightarrow \text{runtime} = \Theta(n^{1.58})$$

# of subproblems reduced!

- Computing multiplication of large integer

$\overline{a_n \cdots a_2 a_1 a_0} = a_n \times 10^n + \cdots + a_1 \times 10^1 + a_0 \rightarrow A(x=10)$

↓

Use algo above to calculate $A(x) \cdot B(x) = C(x)$

↓

Some "digits" would be greater than 9

→ subtract 10 and then add 1 to the next digit

DATE

Master Theorem

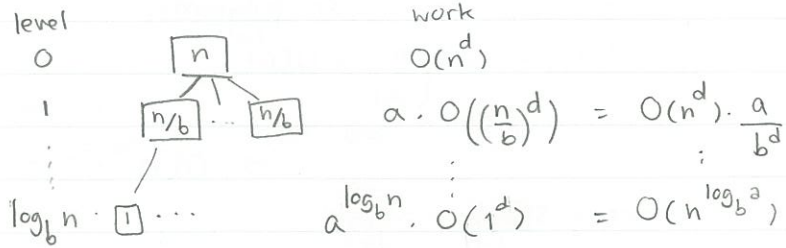
or $\lceil \frac{n}{b} \rceil$ doesn't matter; subproblem is smaller

If $T(n) = a\,T(\lceil \frac{n}{b} \rceil) + T(n^d)$ $\quad a > 0,\ b > 1,\ d \geq 0$

Then:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Proof:



$$\text{Total work} = \sum_{i=0}^{\log_b n} O(n^d)\left(\frac{a}{b^d}\right)^i$$

Case 1: $\frac{a}{b^d} < 1$ $(\rightarrow d > \log_b a)$ or when amount of work decreasing down each level

→ the first term dominates → $O(n^d)$

Case 2: $\frac{a}{b^d} = 1$ $(\rightarrow d = \log_b a)$ or when amount of work is constant at each level

$$= \sum_{i=0}^{\log_b n} O(n^d) = (1 + \log_b n)\,O(n^d) \rightarrow O(n^d \log n)$$

Case 3: $\frac{a}{b^d} > 1$ $(\rightarrow d < \log_b a)$ or when amount of work is increasing down each level

→ the last term dominates → $O(n^{\log_b a})$

last term $\Rightarrow$ $O\left(O(n^d)\left(\frac{a}{b^d}\right)^{\log_b n}\right)$

$= O\left(O(n^d)\frac{a^{\log_b n}}{n^d}\right)$ → $O(n^d)$ is bounded by $k \cdot n^d$ $\Rightarrow$ constant

$= O(n^{\log_b a})$

Example: ◦ Naive divide-and-conquer polynomial mult

$T(n) = 4T\left(\frac{n}{2}\right) + O(n^1)$ (a, b, d)

$1 < \log_2 4$

$\rightarrow O(n^{\log_2 4}) = O(n^2)$

◦ Binary search

$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + O(1)$ ($d = 0$)

$0 = \log_2 1$

$\rightarrow O(n^0 \log n) = O(\log n)$

DATE

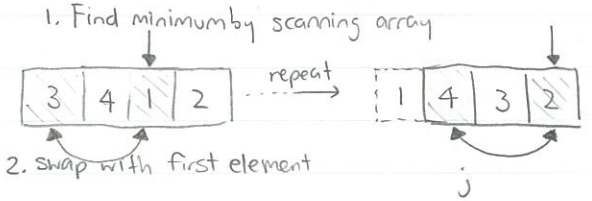
Sorting Problem

Input: Sequence $A[0 \ldots n-1]$

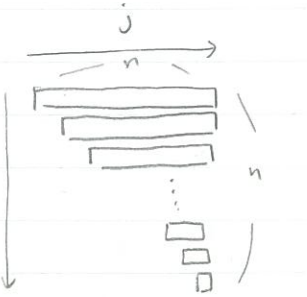
Output: Permutation $A'[0 \ldots n-1]$ of $A[0 \ldots n-1]$ in non-decreasing order

→ an important step of many efficient algorithms

◦ Selection sort




```
def SelSort(A):
    for i in range(n):
        minIndex = i
        for j in range(i+1, n):
            if A[j] < A[minIndex]:
                minIndex = j
    swapping
    A[i], A[minIndex] = A[minIndex], A[i]
```



$\frac{n(n+1)}{2}$ iterations

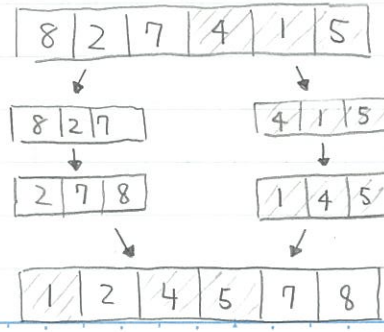
→ Runtime $O(n^2)$ regardless of initial positions

→ sorts in place, requiring constant amount of extra memory

→ to store indices i, j, etc. but doesn't increase with n

◦ Other $O(n^2)$ sort algorithms: insertion sort, bubble sort

◦ Merge sort ← Divide-and-conquer 방식

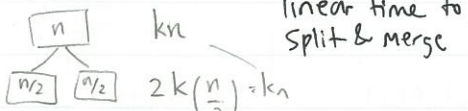


```
def MergeSort(A):
    if n==1: return A
    m = n // 2
    B = MergeSort(A[:m])
    C = MergeSort(A[m:])
    A' = Merge(B,C)
    return A'
```

```
def Merge(B,C): # B&C already sorted
    D = []
    while len(B) > 0 or len(C) > 0:
        if len(B) == 0:
            D.extend(C); break
        if len(C) == 0:
            D.extend(B); break
        if B[0] <= C[0]:
            D.append(B[0])
            B = B[1:]
        else:
            D.append(C[0])
            C = C[1:]
    return D
```

Recurrence:

$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$ ← linear time to split & merge



$a=2, b=2, d=1 \Rightarrow 1 = \log_2 2$

Runtime is $O(n \log n)$.

Lower Bound

- Comparison-based sorting algorithm sorts objects by comparing pairs of them.

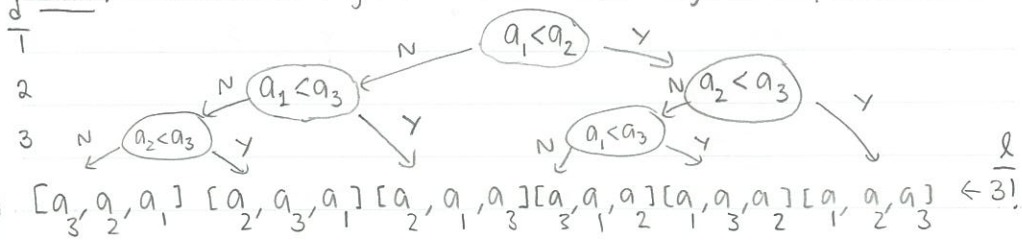
  e.g. selection sort (compare if smaller), merge sort (when merge)

* **Lemma**

Any comparison-based sorting algo performs $\Omega(n \log n)$ comparisons in the worst case scenario.

(i.e. $\exists A$ s.t. algo performs at least $\Omega(n \log n)$ comparisons to sort A.

Proof: Visualize sorting as decision tree e.g. $A = [a_1, a_2, a_3]$



- Number of leaves $\ell = n!$
- Worst case running time is at least depth d
- Given d, max (leaves) $= 2^d \Rightarrow 2^d \geq \ell$ or $d \geq \log_2 \ell$
- Running time is at least $\log_2(n!) = \Omega(n \log n)$

  Because: $\log_2(n!) = \log_2(1 \cdot 2 \cdot \cdots \cdot n) = \log_2 1 + \log_2 2 \cdots \log_2 \frac{n}{2} + \cdots \log_2 n$

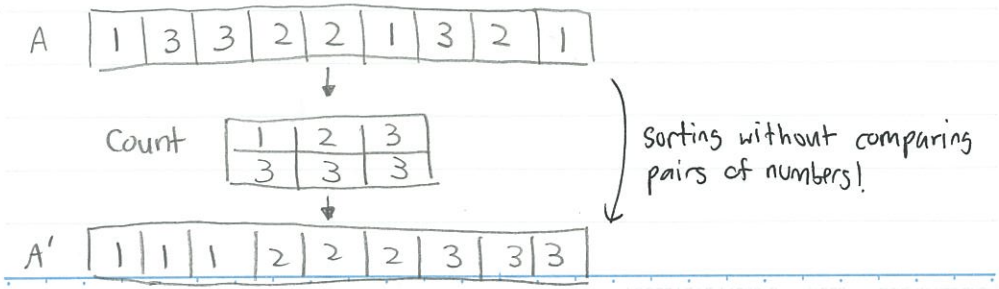
  $\geq \log_2 \frac{n}{2} + \cdots + \log_2 n$ (throw away first half)

  $\geq \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)$. Q.E.D.

→ Among comparison-based sort, merge sort is asymptotically optimal. (no other " " can be asymptotically faster).

* However, if we know something about what we want to sort, can be faster. e.g. if sorting small integers.

- **Count sort** (non-comparison-based)



```
def CountSort(A):          → Assume value is from 0 to M-1
    count = [0]*M
    for i in range(n):
        count[A[i]] += 1
    pos = [0]*M     # storing position of each integer
    for j in range(1:n):
        pos[j] = pos[j-1] + count[j-1]
    for i in range(n):
        A'[pos[A[i]]] = A[i]
        pos[A[i]] += 1
```

if range of possible integers grows no faster than n (bounded from above)

Runtime = O(n+M)

* If M = O(n) then runtime = O(n)

worst-case $O(n^2)$

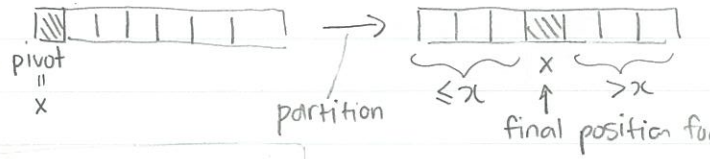
average $O(n \log n)$

↓ quicker than merge sort (usually)

Quick Sort

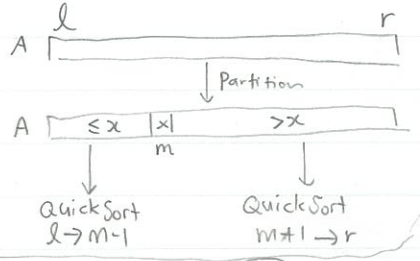
→ comparison-based algorithm, quick and efficient in practice

Idea: "Partition" with pivot element (say A[0]) and put that pivot element into its final position.



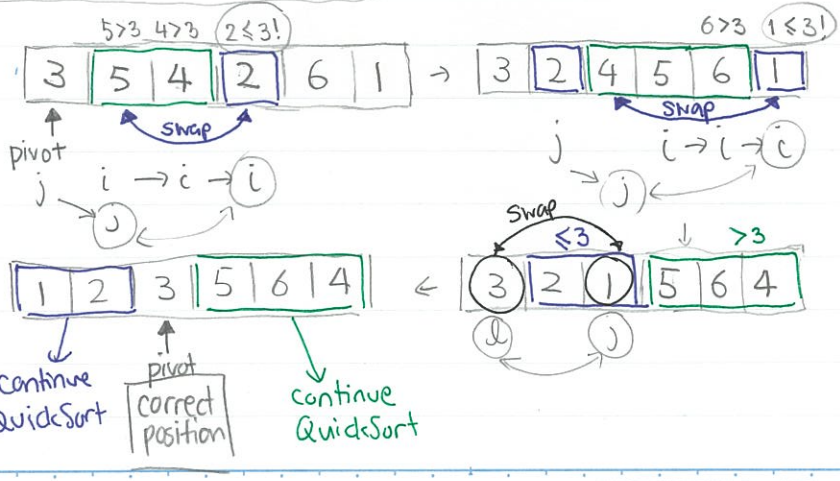

```
def QuickSort(A, l, r):
    if l ≥ r : return          # there's at most one element so return.
    m = Partition(A, l, r)     # return pos m where at this point
                               #   A[m] is in its final position
    QuickSort(A, l, m-1)
    QuickSort(A, m+1, r)
```

we know that A[l→m-1] is ≤ A[m] and A[m+1→r] > A[m] so can sort these two groups independently



Partitioning

· First assume always choose the leftmost element (A[l]) as pivot



```
def Partition(A, l, r):
    x = A[l]    # choose leftmost to be pivot (for now)
    j = l
    for i in range(l+1, r+1):        ← l+1 to r
        if A[i] <= x:    # if smaller than pivot then swap
            j += 1
            A[j], A[i] = A[i], A[j]    # swap
    A[l], A[j] = A[j], A[l]    # put pivot in the right place
    return j    # return pivot position
```

Issue 1: Unbalanced Partition & Runtime

- Worst-case scenario runtime is $O(n^2)$.
  - → when partition, always split into 0 and n-1.

$T(n) = T(n-1) + T(0) + O(n)$

$= n + (n-1) + (n-2) + \dots = \Theta(n^2)$.

- → if unbalanced, still $O(n^2)$

e.g. $T(n) = T(n-5) + T(4) + O(n)$

$\geq n + (n-5) + (n-10) + \dots = \Theta(n^2)$.

- If partition is balanced, runtime is $O(n \log n)$

e.g. $T(n) = 2T(n/2) + O(n) \rightarrow O(n \log n)$.

* Must ensure partition is balanced → random pivot.

even $T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + O(n) \Rightarrow$ still $O(n \log n)$.


```
def RandomQuickSort(A, l, r):
    if l ≤ r : return
    k = random number between l and r
    A[l], A[k] = A[k], A[l]
    m = partition(A, l, r)
    RandomQuickSort(A, l, m-1)
    RandomQuickSort(A, m+1, r)
```

if all elements in A are pairwise different, then

avg runtime = $O(n \log n)$

worst case = $O(n^2)$

→ avg over random number not over the inputs. Regardless of inputs!

Proof

- Runtime is proportional to the number of comparisons made.
- Balanced partition → less number of comparisons
- A given pair of A'[i], A'[j] are compared once (when one of them is selected as a pivot) or not at all,

→ max no of comparisons = $\binom{n}{2} = O(n^2)$ → worst-case.

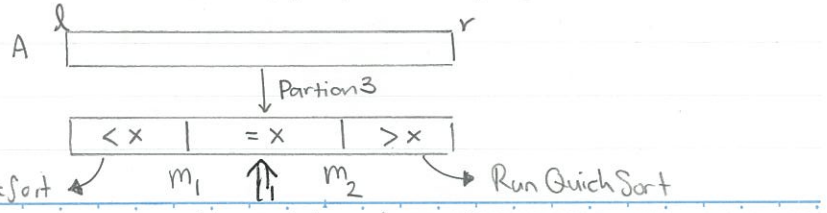
- Using random pivot, A'[i, ..., j] is compared with prob = $\frac{2}{j-i+1}$

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i<j} \frac{2}{j-i+1} \leq 2n\left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}\right) = \Theta(n \log n)$$

Issue 2: Equal elements & 3-way Quick sort

- If all elements are equal, QuickSort very slow as always partition into size n-1 and 0. (unbalanced partition) → 3-way Partition!

instead of $m = \text{Partition}(A, l, r)$, use $(m_1, m_2) = \text{Partition3}(A, l, r)$



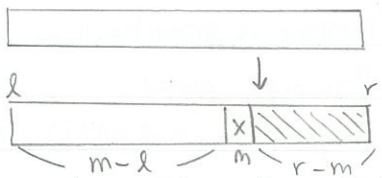
No need to sort again!

Issue 3: Space complexity & Tail Recursion Elimination

- Since we're always sorting array A, no additional space required
- However, when we make recursive call, we incur stack space.
- Since recursive call on left & right parts are independent (not order-sensitive), we can ensure worst case space requirement to be $O(\log n)$ by using Tail Recursion Elimination.

(* Space requirement largely depends on the depth of recursive call)

```
def QuickSort(A, l, r):
    while l < r:
        m = Partition(A, l, r)
        if (m-l) < (r-m):
            QuickSort(A, l, m-1)
            l = m+1
        else:
            QuickSort(A, m+1, r)
            r = m-1
```



do recursive call on shorter array first!

Call $n \to \leq \frac{n}{2} \to \leq \frac{n}{4} \to \dots$

depth $\leq \log_2 n$

$\to O(\log n)$ space req.

Issue 4: Reproducibility (Deterministism).

- Using random number → not reproducible → use simple heuristis instead (e.g. median of 1^{st}, middle, last element)
  - → if recursion depth exceeds certain threshold then switch to heap sort → worst case is $O(n \log n)$!
    - ↳ $O(n \log n)$

Dynamic Programming (動的計画法)

Change Problem

Input: Integer money and coin denominations $c_1, c_2, \dots, c_d$

Output: Minimum # of coins that changes money

- Greedy algorithm → incorrect answer!

```
while money > 0:
    money -= largest c_i that doesn't exceed money
    coin += 1
```

e.g. change money = 40¢

US coins = 1¢, 5¢, 10¢, 25¢ → 25+10+5 → 3 ✓

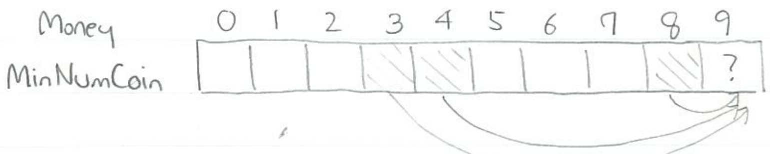
Tanzania Coins = 1¢, 5¢, 10¢, 20¢, 25¢ → 25+10+5 → 3 > 2 ← 20+20

Not Optimal

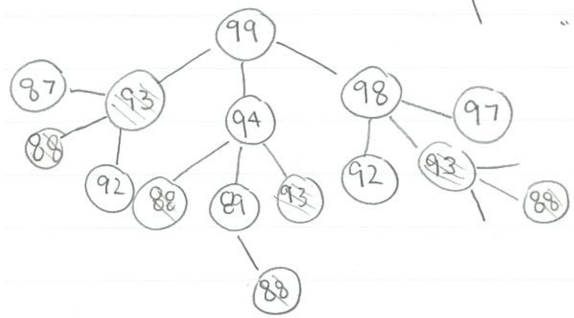
- Recursive algorithm → always correct but takes forever to compute!

e.g. money = 9, c = [1, 5, 6]

Right → Left



$$\text{MinNumCoin(money)} = \min \begin{pmatrix} \text{MinNumCoin(money} - c_1) \\ \text{" (money} - c_2) \\ \vdots \\ \text{" (money} - c_d) \end{pmatrix} + 1$$



Repeat computation too many times

↓

Cannot compute in reasonable amount of time!

NO.

DATE

プログラミングとなんら関係ない勉強

- Dynamic Programing algorithm
  → instead of making time-consuming recursive calls,
  we simply look-up the values that have already been computed.

C = [1, 5, 6]

money	0	1	2	3	4	5	6	7	8	9
MinNumCoins	0	1	2	3	4	1	1	2	3	4

Left → Right

$= \min\begin{pmatrix}\text{MinNumCoins}(5-1),\\ \text{MinNumCoins}(5-5)\end{pmatrix} + 1$

$= \min(4, 0) + 1 = 1$

$\min(1, 1, 0) + 1 = 1$

$\min(3, 4, 3) + 1 = 4.$

[9-1] [9-5] [9-6]

→ Runtime = $O(md)$ (for each cell up to m, must try changing with d denominations).

money

NO. 14

DATE

The Alignment Game & Edit Distance (String Comparison)

Definition: Alignment score (given two strings) is

matches $- \mu \cdot$ # mismatches $- \sigma \cdot$ # indel

e.g matching gene sequence (to see if it's similar) (insertion and deletion)

match (+1) insertion deletion mismatch

ATGTTATA →	A	T	-	G	T	T	A	T	A
ATCGTCC →	A	T	C	G	T	-	C	-	C
	+1	+1	$-\sigma$	+1	+1	$-\sigma$	$-\mu$	$-\sigma$	$-\mu$

$= 4 - 3\sigma - 2\mu.$

common subsequence

- Longest Common Subsequence ⇒ maximizing length of common subsequence ⇒ maximizing alignment score with $\mu = \sigma = 0$.
  ⇒ Minimizing edit distance when allowing insertion & deletion only

* Min(edit distance) ⟺ Max(alignment score)

Edit distance ⇒ Minimum # of operations (insertion, deletion, substitution of symbol) to transform one string to another

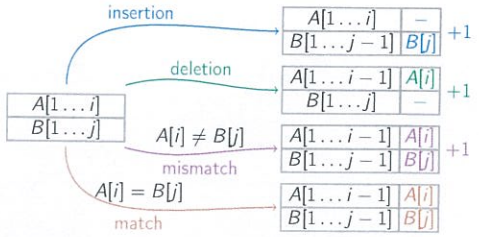
e.g. EDITING, DISTANCE ⇒

E	D	I	-	T	I	N	G	-	
-	D	I	S	T	A	N	C	E	

edit distance = 5, delete E, insert S, substitute I ⟺ A, substitute G ⟺ C, insert E

- Computation
  - input Given A[1, ..., n] and B[1, ..., m]
  - output edit distance & optimal alignment A[1, ..., i], B[1, ..., j]

Let $D(i,j)$ be edit distance of $A[1 \ldots i]$ and $B[1 \ldots j]$ then
(i-prefix of A, j-prefix of B)



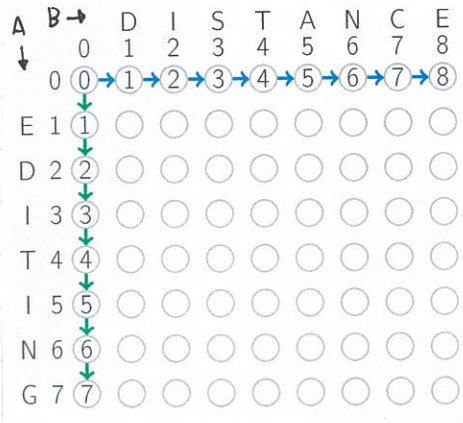
← 4 possibilities

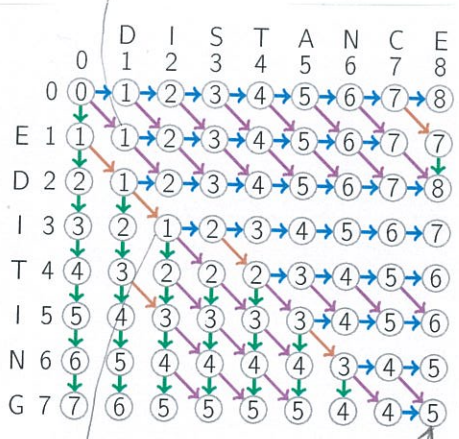
$$D(i,j) = \min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+1 & \text{if } A[i] \neq B[j] \\ D(i-1,j-1) & \text{if } A[i] = B[j] \end{cases}$$

→ compute recursively

since A[1] ≠ B[1],
$\min(D(1,0)+1, D(0,1)+1, D(0,0)+1)$
↓ 1, ↓ 1, ↓ 0

Initialization





since A[3] = B[2] then
$\min(D(3,1)+1, D(2,2)+1, D(2,1))$
↓ 2, ↓ 2, ↓ 1

edit distance

EditDistance($A[1 \ldots n], B[1 \ldots m]$)

$D(i,0) \leftarrow i$ and $D(0,j) \leftarrow j$ for all i,j

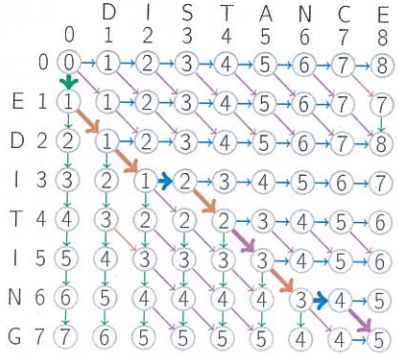
```
for j from 1 to m:
  for i from 1 to n:
    insertion ← D(i, j − 1) + 1
    deletion ← D(i − 1, j) + 1
    match ← D(i − 1, j − 1)
    mismatch ← D(i − 1, j − 1) + 1
    if A[i] = B[j]:
      D(i, j) ← min(insertion, deletion, match)
    else:
      D(i, j) ← min(insertion, deletion, mismacth)
return D(n, m)
```

Use backtracking to find the optimal alignment

↓

OutputAlignment(i,j)

```
if i = 0 and j = 0:
  return
if i > 0 and D(i, j) = D(i − 1, j) + 1:
  OutputAlignment(i − 1, j)
  print A[i] / –
else if j > 0 and D(i, j) = D(i, j − 1) + 1:
  OutputAlignment(i, j − 1)
  print – / B[j]
else:
  OutputAlignment(i − 1, j − 1)
  print A[i] / B[j]
```



E	D	I	–	T	I	N	–	G
–	D	I	S	T	A	N	C	E

Knapsack Problem (2)

- **Fractional Knapsack** → Greedy algorithm (can take fraction of items.)
- **Discrete knapsack**
  - with repetitions (unlimited quantities)
  - without repetitions (0-or-1)

  } Greedy algo doesn't work! Dynamic Programming.

(31)

W = 10

	v = 30	14	9	16
	w = 6	3	2	4
v/w	5	$4\frac{2}{3}$	$4\frac{1}{2}$	4

taking element with max(v/w) is not a safe move!

w/ rep optimal →

30	9	9
6	2	2

→ (48)

30	14
6	3

→ ~~44~~ not optimal.

w/o rep optimal →

30	16
6	4

→ (46)

* Assume w_i, v_i, W are non-negative integers.

With repetitions.

value(w) = max value given weight w

if this is optimal	v_i / w_i
this is optimal too	$W - w_i$

W

$$\text{value}(w) = \max_{i:\, w_i \le w} \left(\text{value}(w - w_i) + v_i \right)$$

```
def Knapsack(W):
    value = [0, 0, ..., 0]   (0 → W)
    for w from 1 → W:
        for i from 1 → n:
            if w_i ≤ W:
                val = value(w - w_i) + v_i
                if val > value(w): value(w) = val
    return value(W)
```

Iterative

→ two nested loops so O(nW) (but not polynomial time! see later)

Iterate from left to right

W:	0	1	2	3	4	5	6	7	8	9	10
Value	0	0	9	14	18	23	30	32	39	44	48

+16, +14, +9

max(9+9, 0+14, 0+16) = 18

max(18+30, 30+16, 32+14, 39+9) = 48

Without repetitions

If this is optimal | w_n W using item 1, 2, ..., n

⇩

then this is optimal $W - w_n$ using only item 1, 2, ..., n-1

$$\text{value}(w, i) = \max \left(\underbrace{\text{value}(w - w_i, i-1) + v_i}_{i^{th} \text{ is used}}, \underbrace{\text{value}(w, i-1)}_{i^{th} \text{ isn't used}} \right)$$

i \ W	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	9	14	14	23	30	30	39	44	44
4	0	0	9	14	16	23	30	30	39	44	46

max(value(0, 3) + 16, value(4, 3)) = 16 (0, 14)

i = 1 → 30 / 6

i = 2 → 14 / 3

i = 3 → 9 / 2

i = 4 → 16 / 4

using only 30 / 6

using only 30 / 6 and 14 / 3

Reconstruction ⇒

i	1	2	3	4
	1	0	0	1

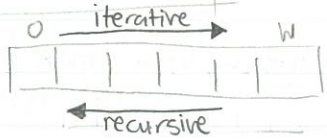
46 = 30 + 16 (4th item was used.)

30 → 30 > 14 + 9

6	4

```
def KnapSack2(W):
    value = [ 0 ... W
              n ... ]
    for i from 1 → n:
        for w from 1 → W:
            value(w,i) = value(w,i-1)
            if w_i ≤ w
                val = value(w-w_i, i-1) + v_i
                if val > value(w,i): value(w,i) = val
    return value(W,n)
```

↗ same two-nested loops as before so $O(nW)$ but not polynomial time.



Considerations

- Iterative vs. Recursive
  - iterative: bottom-up approach (solving small → big problem)
  - recursive: top-down approach (solving big → small problem)
    - ↳ not efficient if computing solutions for the same subproblem multiple times → use memoization!

```
def Knapsack3(W):
    if W is in hash table: return value(W)
        ⋮
    val = Knapsack3(W-w_i) + v_i
        ⋮
    insert value(W) into hash table with key W
return value(W)
```

Which one faster?

- If all subproblems must be solved → iterative!
  (as recursive algo incurs overhead e.g. checking hash table)

- if need not solve all subproblems, recursive can be faster as doesn't need to solve unnecessary subproblems.

Discrete Knapsack running time is exponential.

- $O(nW)$ looks like polynomial time ... it's not. Pseudo-polynomial
  - → W is not polynomial in the length of input
    e.g. for $W = 1{,}000{,}000{,}000{,}000$, it only takes 40 bits to represent this number so input size = 40.
    but the number of operations is close to 2^{40} operations!
  - → Running time is $O(n \cdot 2^{\log W})$ → exponential time
    (log W: # of bits in W)

DP Example3: Placing Parentheses.

How to place parentheses in an expression $5-8+7\times4-8+9$ to maximize its value?

Ans: $5-((8+7)\times(4-(8+9))) = 200$.

Brute-force method: try all permutations of arithmetic operations
e.g. in the example, 5 operations → $5! = 120$ possible permutations $O(n!)$
$n!$ grows extremely large very quickly! → use DP.

Input given digits $d_1, \dots, d_n$ and operations $op_1, \dots, op_{n-1} \in \{+, -, \times\}$
Output the order of these operations that maximizes value.

Assuming ⊗ is the last operation → we need to know the optimal values of subproblems $(5-8+7)$ and $(4-8+9)$.
Since we have ⊗ and ⊖, need to keep track of both maximum and minimum values of subproblems.

NO.

DATE . .

e.g. $(5-8+7) \times (4-8+9) \rightarrow$ max $= -10 \times -13 = 130$.

max is 4 ↙ max is 5 ↓ not $4 \times 5 = 20$.

min is $5+(8+7) = -10$ min is $4-(8+9) = -13$

(*) · max of subprob × max of subprob ≠ max of (subprob × subprob)

Algorithm.

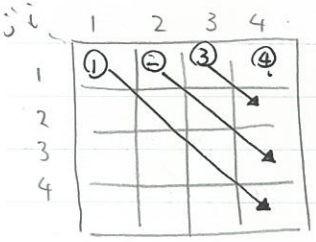
Let $M(i,j) = \max \begin{pmatrix} M(i,k)\ op_k\ M(k+1,j) \\ M(i,k)\ op_k\ m(k+1,j) \\ m(i,k)\ op_k\ M(k+1,j) \\ m(i,k)\ op_k\ m(k+1,j) \end{pmatrix}$ → max & min of subexpressions $d_i\ op_i \cdots op_{j-1}\ d_j$

$m(i,j) = \min \begin{pmatrix} M(i,k)\ op_k\ M(k+1,j) \\ M(i,k)\ op_k\ m(k+1,j) \\ m(i,k)\ op_k\ M(k+1,j) \\ m(i,k)\ op_k\ m(k+1,j) \end{pmatrix}$

```
def MinAndMax(i,j):
    min, max = +∞, -∞
    [for k from i → j-1:
        ⋮
        min, max = ...
    return (min, max)
```

```
def Parentheses(d_1 op_1 ... d_n):
    for i from 1 → n:
        m(i,i), M(i,i) = d_i, d_i
    [for s from 1 → n-1:
        [for i from 1 → n-s:
            j = i+s   # diagonal cells
            m(i,j), M(i,j) = MinAndMax(i,j)
    return M(1,n)   ← solution
```

Order of calculation



3 nested loops → $O(n^3)$

Divide-and-conquer → subproblems are disjoint

Dynamic Programing → " usually overlapping.

NO.

DATE . .