Attributes of Algorithm: Correctness, Efficiency

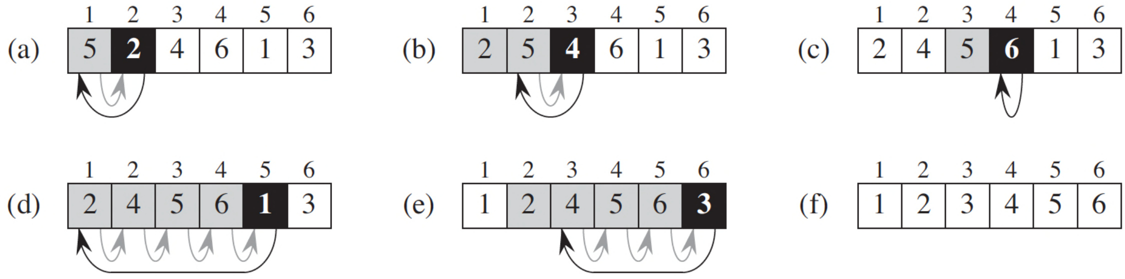
Correctness: For any given input, it halts with the correct output.

↳ 如果 Algorthm A Algorithm B => 都解决了问题

怎么选择? → {Ease of understanding, Elegance, Efficiency (space and time)}

Order of an Algorithm:

Insertion Sort: Sequence of n numbers → $\boxed{\text{insertion sort}}$ → Sorted sequence

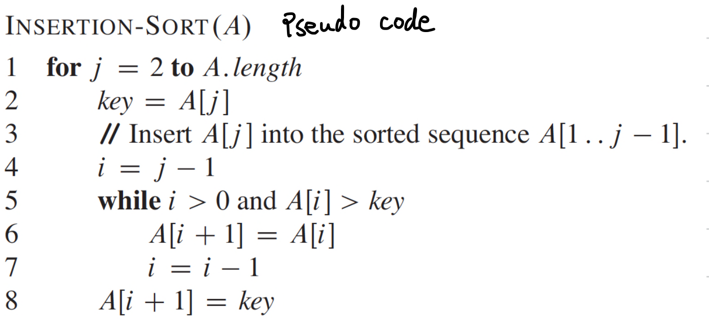


(a) 1 2 3 4 5 6 | 5 2 4 6 1 3
(b) 1 2 3 4 5 6 | 2 5 4 6 1 3
(c) 1 2 3 4 5 6 | 2 4 5 6 1 3
(d) 1 2 3 4 5 6 | 2 4 5 6 1 3
(e) 1 2 3 4 5 6 | 1 2 4 5 6 3
(f) 1 2 3 4 5 6 | 1 2 3 4 5 6

*从第2个element开始 sort

↳ 1.当ptr 指向这个element, 用另一个 loop, 去遍历这个 element 之前的 sorted array

2. 然后把 element 插入一个比它大的 element 之前



INSERTION-SORT(A) Pseudo code

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

→ 对不同问题所取的 input size 不一样

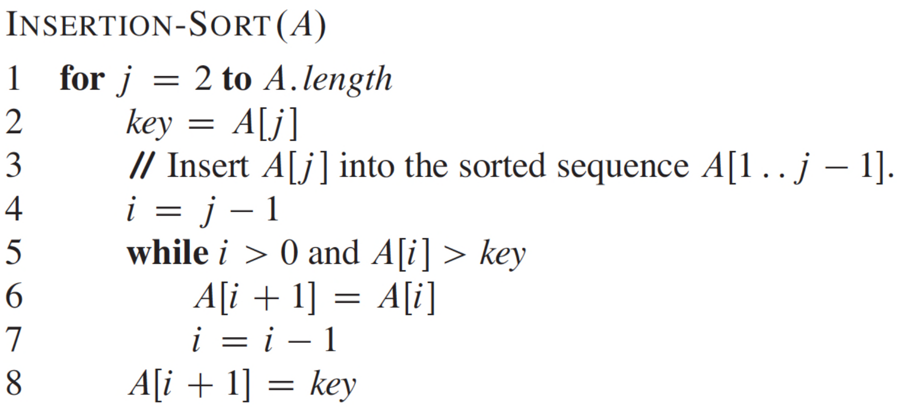
* Analyze insertion-sort: 时间 depends on {The input size (array size n for sorting), How sorted the input array is}

↓

Running time: The running time of an algorithm on a particular input is the number of primitive operations or "step" executed 总共有多少个 step

对于 Pseudo code 来说: ith 行用时 C_i

几个 notation:
- $n = A.length$
- t_j denote the number of times while loop is executed
- Cost 指的是运行这一行所使用的时间，$T(n)$ 代表总的 running time



```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Cost	times
c_1	n
c_2	$n-1$
c_4	$n-1$
c_5	$\sum_{j=2}^{n} t_j$
c_6	$\sum_{j=2}^{n} (t_j - 1)$
c_7	$\sum_{j=2}^{n} (t_j - 1)$
c_8	$n-1$

还要检查 condition 所以 c_1 比 c_2 多一次

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\sum_{j=2}^{n} t_j\right) + c_6\left(\sum_{j=2}^{n}(t_j-1)\right) + c_7\left(\sum_{j=2}^{n}(t_j-1)\right) + c_8(n-1)$$

Best case 的话，就是 input 的 array 是处理好的：

1，2，3，4 行後
5 行并不会进入 loop → 6，7 不执行
5 $(n-1)$　　8 $(n-1)$

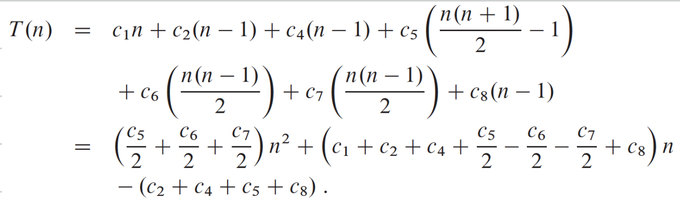
$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \Rightarrow \text{Linear function}$$

Worst case：本来就会都是反的，所以在 5 行中的 loop 每次都会检查遍整个 sorted array

对第 5 行：$t_j = j$ ： $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ ／ $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$

$$t_j = j$$

$$= 2+3+4+5 \cdots + n$$
$$= 2+3+4+ \cdots + n-2 + n-1 + n$$
$$= 2+(n-1) + 3+(n-2) + \cdots + (n+1) - 1$$
$$= \frac{n}{2} \times (n+1) = \frac{n(n+1)}{2} - 1$$



$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

⟹ quadratic function

为什么一般选用 worst case analysis：提供了一个 upper bound for any given input

Order of growth：简化的 running time，在这个里面，不仅实际的时间不重要，就连 abstract 的 "cost" 也不重要

↳ 1. 只考虑 leading term，$\left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2$ for insertion sort

2、把常数系数 drop 掉 ⟶ $n^2 \to O(n^2)$: worst case running time

Θ -notation

实际的 T(n)

$\Theta(g(n)) = \{f(n)$: there exist positive costants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$

g(n) grows at ~~a~~ rate the same as g(n)

Analyzing computational complexity

- For a sequence of statements, consider the complexity of the most complex thing in the sequence
- ✱ For a loop, consider the complexity of the body of the loop multiplied by the number of times the loop executes
- ✱ For a conditional statement, consider the complexity of whichever of the two alternatives has the higher complexity

How exactly?

- Identify the fundamental step that is executed the most often
- Write down a function that relates the number of times that step executes to the size of the input $T(n)$
- Simplify that function as much as possible by discarding smaller terms and constant coefficients
- What remains is the complexity of the algorithm/code

* $\Theta(g(n))$ 实际上是一个 set，$f(n)$ 是其中的一员

$O(g(n))$ ↑

$\Omega(g(n))$

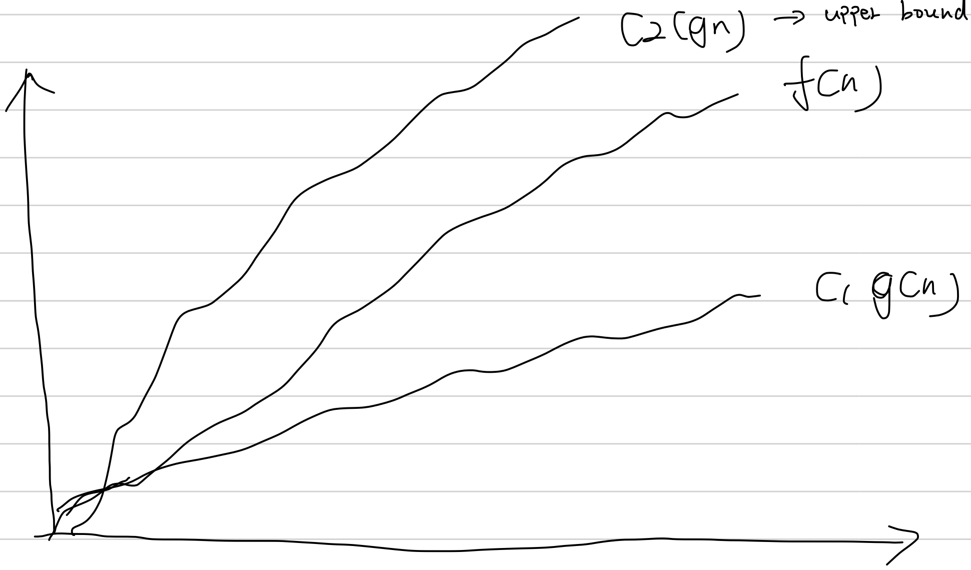
Θ-notation

实际的 $T(n)$

$\Theta(g(n)) = \{f(n)$: there exist positive costants c_1, c_2, and n_0 such that

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$

$g(n)$ grows at ~~a~~ rate the same as $g(n)$



→ upper bound

所以 $g(n)$提供了一个 tight bound

→ lower bound

比如 $\frac{1}{2}n^2 - 3n$ => $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$

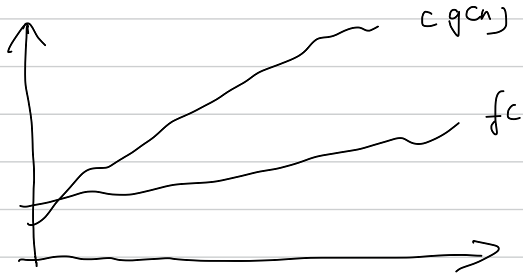
$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$

O-notation

实际的 $T(n)$

$O(g(n)) = \{f(n)$: there exist positive costants c and n_0 such that

$0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0\}$



=> 提供了一个 upper bound

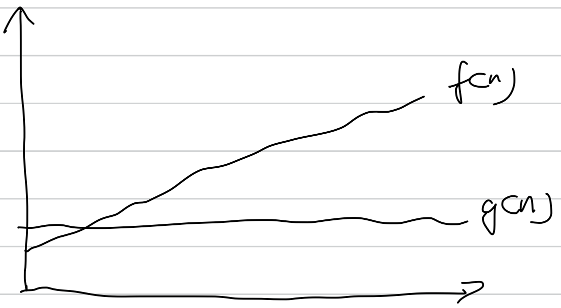
=> if O-notation bounds the worst case running time of an algorithm, it bounds the running time of the algorithm on every input ??

Ω-notation

实际的 $T(n)$

$\Omega(g(n)) = \{f(n)$: there exist positive costants c and n_0 such that

$0 \leq c\,g(n) \leq f(n)$ for all $n \geq n_0\}$



=> 提供了一个 lower bound

课本补充:

* $\Theta(g(n))$ requires that every member $f(n) \in \Theta(g(n))$ be asymptotically nonnegative

* asymptotically nonnegative : $f(n) \geq 0$ whenever n is sufficiently large

* In general, for any polynomial $p(n) = \sum_{i=0}^{d} a_i n^i$ where the a_i are constant and $a_d > 0$ we have $p(n) = \Theta(n^d)$

* Note $\Theta(g(n))$ implies $O(g(n))$ → $\underbrace{\Theta(g(n))}_{} \subseteq \underbrace{O(g(n))}_{}$

⇓

两个 set → 属于

* the $\Theta(n^2)$ bound on the worst-case running time of insertion sort, however doesn't imply a $\Theta(n^2)$ bound on the running time of insertion sort on every input.

→ 因为 $\Theta(g(n))$ 是一个 tight bound, 而 insertion sort 并不是处处都被 $\Theta(n^2)$ bound 起

* When we say that the running time of an algorithm is $\Omega(g(n))$

⇓

No matter what particular input of size n is chosen for each value of n, the value is at least a constant times $g(n)$

⇓

lower bound for even best case scenario

Asymptotic notation in equations and inequalities

⇒① 可以直接把 asymptotic notation 用在式子中

$2n^2 + 3n + 1 = 2n^2 + \underbrace{\Theta(n)}_{}$

↓ $f(n)$ which in $\Theta(n)$

→ eliminate inessential detail and clutter in an equation

② $2n^2 + \Theta(n) = \Theta(n^2)$

No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid

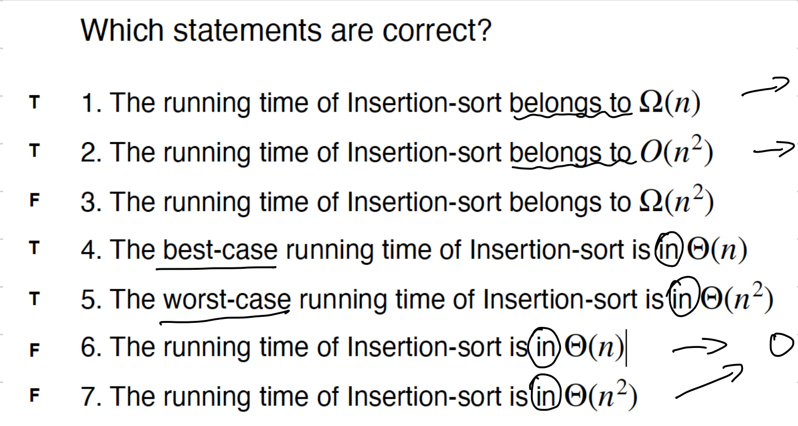
→ $2n^2 + \Theta(n) = \Theta(n^2)$

for any function $f(n) \in \Theta(n)$, there is some function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$

甚至还可以把①,②串在一起: $2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$

Theorem: $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

⇒ 只要记住 $O(n)$ 是 worst case, $\Omega(n)$ 是 best case 就好了



Which statements are correct?

T 1. The running time of Insertion-sort belongs to $\Omega(n)$ → best case

T 2. The running time of Insertion-sort belongs to $O(n^2)$ → worst case

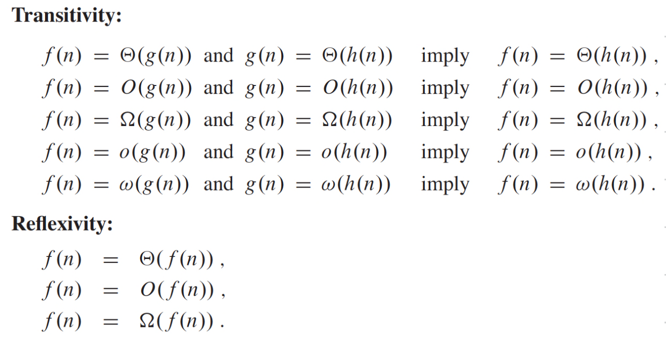
F 3. The running time of Insertion-sort belongs to $\Omega(n^2)$

T 4. The best-case running time of Insertion-sort is in $\Theta(n)$

T 5. The worst-case running time of Insertion-sort is in $\Theta(n^2)$

F 6. The running time of Insertion-sort is in $\Theta(n)$ → $O(g(n)) \neq \Omega(g(n))$

F 7. The running time of Insertion-sort is in $\Theta(n^2)$

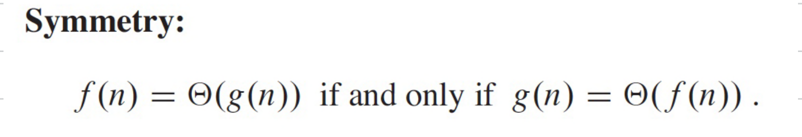


Transitivity:

$$
\begin{aligned}
f(n) = \Theta(g(n)) &\text{ and } g(n) = \Theta(h(n)) &\text{ imply } f(n) = \Theta(h(n)),\\
f(n) = O(g(n)) &\text{ and } g(n) = O(h(n)) &\text{ imply } f(n) = O(h(n)),\\
f(n) = \Omega(g(n)) &\text{ and } g(n) = \Omega(h(n)) &\text{ imply } f(n) = \Omega(h(n)),\\
f(n) = o(g(n)) &\text{ and } g(n) = o(h(n)) &\text{ imply } f(n) = o(h(n)),\\
f(n) = \omega(g(n)) &\text{ and } g(n) = \omega(h(n)) &\text{ imply } f(n) = \omega(h(n)).
\end{aligned}
$$

Reflexivity:

$$
\begin{aligned}
f(n) &= \Theta(f(n)),\\
f(n) &= O(f(n)),\\
f(n) &= \Omega(f(n)).
\end{aligned}
$$



Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).$$

$\Theta(f(n) + g(n))$ ⇒ there exist constant c_1, c_2, n_0 such that

$$0 \le c_1(f(n) + g(n)) \le T(n) \le c_2(f(n)+g(n))$$

$\max(f(n), g(n))$ $\qquad c_1 \le \frac{T(n)}{f(n)+g(n)} \le c_2$

$\downarrow$ ⇒ $0 \le c_1 \le \frac{f(n)}{f(n)+g(n)} \le c_2$

$f(n) > g(n)$

if $\max(f(n), g(n)) = f(n)$:

3.1-1 : ==

$\exists\, n_1, n_2$: $f(n) \ge 0$, for $n > n_1$

$g(n) \ge 0$ for $n \ge n_2$

let $n_0 = \max(n_1, n_2)$, for $n > n_0$:

$$f(n) \le \max(f(n), g(n))$$
$$g(n) \le \max(f(n), g(n))$$
$$f(n) + g(n) \le 2\max(f(n), g(n))$$
$$(f(n) + g(n)) \times \tfrac{1}{2} \le \max(f(n), g(n))$$
$$\max(f(n), g(n)) \le f(n) + g(n)$$
$$\Downarrow$$
$$\max(f(n), g(n)) \le f(n) + g(n) \le \tfrac{1}{2}\max(f(n), g(n))$$
$$\Downarrow$$
$$\tfrac{1}{2}(f(n) + g(n)) \le \max(f(n), g(n)) \le f(n) + g(n)$$
$$\Downarrow$$
$$c_1 = \tfrac{1}{2}\ ,\ c_2 = 1$$

Complexity of Recursive Algorithm

Recursive Algorithm 又被称为 divide and conquer algorithms

↳ divide, conquer, combine => Merge Sort!

Merge Sort: Divide the n-element sequence to be sorted into two subsequences of $\frac{n}{2}$ elements each, sort the two subsequences recursively using merge sort, then merge the two sorted arrays.

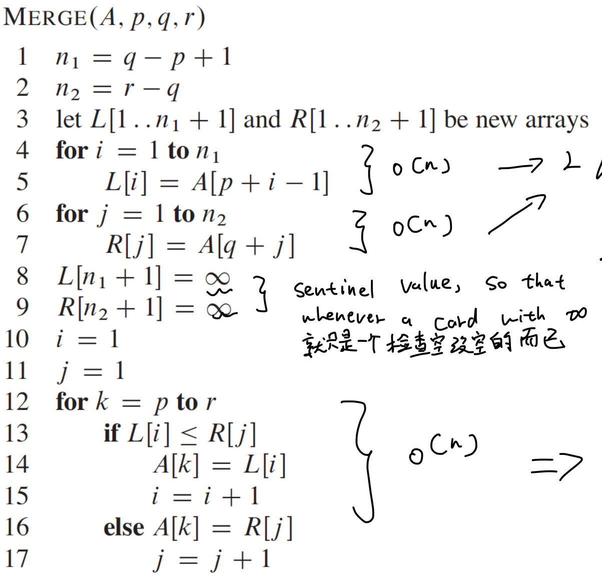
↳ 最关键的操作是 merge (A, p, q, r) => A是一个 array

A[p ... q] 是一个 sorted array

A [q+1, r] 是另一个 sorted array

⇓ return

sorted Array A(p − r)



```
MERGE(A, p, q, r)
 n_1 = q − p + 1
 n_2 = r − q
 let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 for i = 1 to n_1
     L[i] = A[p + i − 1]
 for j = 1 to n_2
     R[j] = A[q + j]
 L[n_1 + 1] = ∞
 R[n_2 + 1] = ∞
 i = 1
 j = 1
 for k = p to r
     if L[i] ≤ R[j]
         A[k] = L[i]
         i = i + 1
     else A[k] = R[j]
         j = j + 1
```

} O(n) → L /R 中存入原 A 前半 / 后半的内容

} O(n)

} sentinel value, so that whenever a card with ∞ is exposed 就只是一个检查空没空的而已

=> Merge 的 running time $T(n) \subseteq O(n)$

} O(n) =>

一个一个对比, min(L[i], R[j])

L[ ]

R[ ]

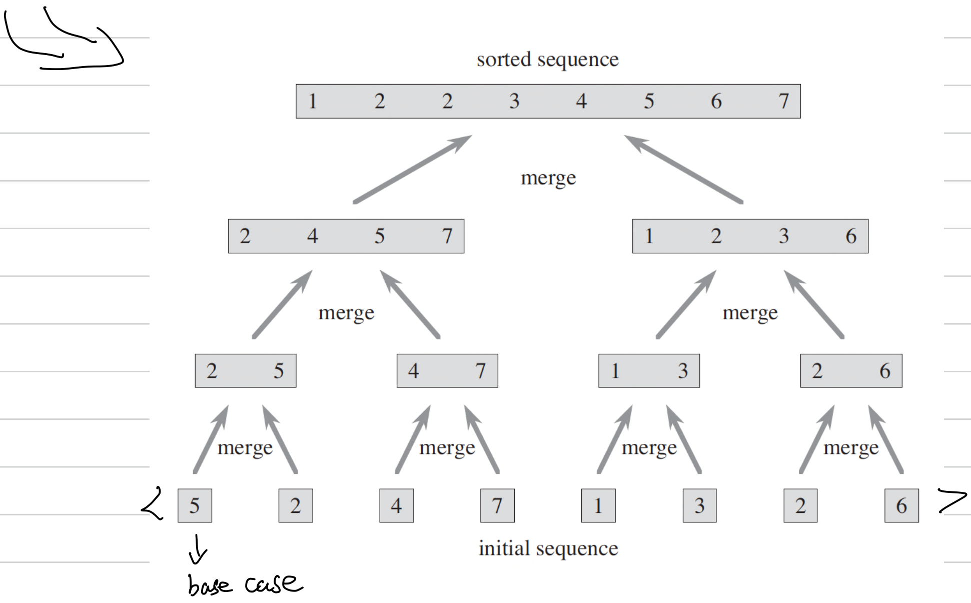
A [ ]

Merge Sort 的 Pseudo code

```
MERGE-SORT(A, p, r)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Analyzing divide and conquer algorithm

* 含有recursive call 的 algorithm 的 running time 被称为 recurrence equation or recurrence = running time on a problem of size n



* Suppose that the division of the problem yields a sub problems, each of which is $\frac{1}{b}$ the size of the original (It takes time $T(\frac{n}{b})$ to solve one sub problem of size $\frac{n}{b}$), $D(n)$ time to divide the problem into sub problems, $C(n)$ to combine the solutions to the subproblems.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c, \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{otherwise} \end{cases}$$

对于 Merge Sort 来说

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(\frac{n}{2}) + 1 + \Theta(n) = \underbrace{2T(\frac{n}{2})}_{\text{2个 subproblems}} + \Theta(n) & \text{if } n > 1 \end{cases}$$

每个subproblem解决 $\frac{n}{2}$ size

Easier Example:

```
A(n):  if n>=1:                         C1
           print n          C2   ?       C2
           A(n-1)           T(n-1)
```

$$T(n) = \begin{cases} C_1 & \text{for } n = 0 \\ \underbrace{C_2}_{\text{可以看作是“当前”的 print } n} + T(n-1) & \text{for } n \geq 1 \end{cases}$$

How to transform the recurrence relation into a closed-form formula?

Substitution Expansion

Substitution

$$\begin{cases} T(0) = C_1 \\ T(1) = C_2 + T(0) = C_2 + C_1 \\ T(2) = C_2 + T(1) = C_2 + C_2 + C_1 \\ T(3) = C_2 + T(2) = C_2 + C_2 + C_2 + C_1 \\ \dots \\ T(n) = C_2 \times n + \underbrace{C_1}_{\text{base case}} \end{cases}$$

$$\begin{cases} T(0) = C_1 \\ T(n) = C_2 \times n + C_1 \end{cases} \Rightarrow \Theta(n)$$

Second Example:

```
B(n):
    if n>=1:
        for i in range(n):
            print i
        B(n-1)
```

$$T(n) = \begin{cases} C_1 & \text{if } n = 0 \\ \underbrace{C_2}_{n^2} + T(n-1) & \text{if } n >= 1 \end{cases}$$

$$\begin{cases} T(0) = C_1 \\ T(1) = 1 + T(n-1) = n + T(0) = n + C_1 \\ T(2) = 2 + T(n-1) = n + T(1) = n + n + C_1 \\ T(3) = 3 + T(n-1) = n + T(2) = n + n + n + C_1 \\ \dots \\ T(n) = n \times n + C_1 = n^2 + C_1 \end{cases}$$

$$\begin{cases} T(0) = C_1 \\ T(n) = n^2 + C_1 \end{cases} \Rightarrow O(n^2)$$

Substitution method:

1. Guess the form of the solution
2. Use mathematical induction to find the constants and show that the solution

Example :

B(n) :

if $n \geq 1$: C_1

for i in range (n) : C_2

print i C_3

B (n-1) 可以用 T(n-1)表示了

$$T(n) = \begin{cases} C_1 & \text{if } n = 0 \\ C_2 + C_3 \times (n) + T(n-1) & \text{if } n \geq 1 \end{cases}$$

$T(0) = C_1$

$T(1) = C_2 + C_3 + T(n-1) = C_2 + C_3 \times 1 + C_1$

$T(2) = C_2 + C_3 \times 2 + T(n-1) \longrightarrow = \underline{C_2} + \underline{C_3 \times 2} + \underline{C_2} + \underline{C_3 \times 2} + C_1$

. . . . .

$T(n) = C_2 n + C_3 \times \left(\underbrace{\sum_{i=1}^{n} i}_{} \right) + C_1$

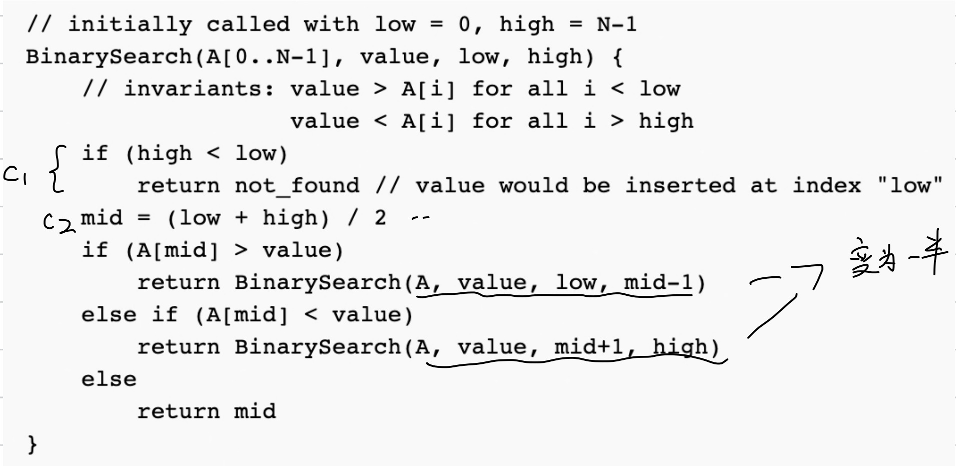
$1+2+3+4+ \cdots + n-2 + n-1 + n$

$= (1+n) + (2+n-1) + (3+n-2) \cdots\cdots$

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

$\longrightarrow T(n) = C_2 n + C_3 \times \frac{n(n+1)}{2} + C_1 \implies \Theta(n^2)$

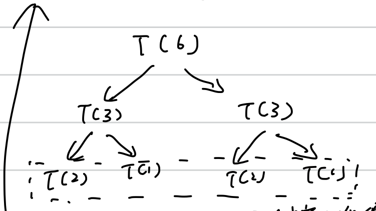
Example: Binary Search



```
    // initially called with low = 0, high = N-1
    BinarySearch(A[0..N-1], value, low, high) {
        // invariants: value > A[i] for all i < low
                       value < A[i] for all i > high
C1 {    if (high < low)
            return not_found // value would be inserted at index "low"
   C2 mid = (low + high) / 2 --
        if (A[mid] > value)
            return BinarySearch(A, value, low, mid-1)
        else if (A[mid] < value)
            return BinarySearch(A, value, mid+1, high)
        else
            return mid
    }
```

变为一半

老师的判断是基于一个理想的情况，即 input $n = 2^k$ for $k = 0, 1, 2, 3 \cdots$ 对于 binary search / merge sort



有这个 assumption

└→这一行的内容似乎很难用一个公式 summarize 出来, so for the sake of simplicity

课中证明了这个 assumption 并不会影响实际的 complexity 的估测

Base case :

当 n=1 的时候， $T(1) = C_1$ (只执行了判断这个值是不是 value 的操作)

// Every input size is 2^k for $k = 0, 1, 2, 3, 4 \cdots$

$a^x = N$

$x = \log_a N$

$T(2) = C_2 + T(\frac{n}{2}) = C_2 + T(1) = C_2 + C_1$

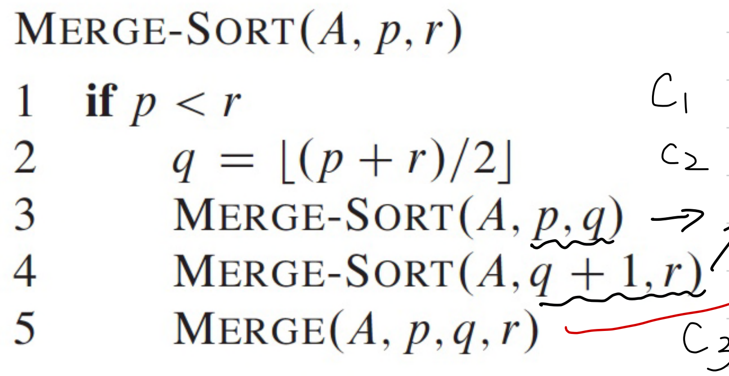
$T(4) = C_2 + T(\frac{n}{2}) = C_2 + T(2) \longrightarrow = C_2 + C_2 + C_1$

$T(8) = C_2 + T(\frac{n}{2}) = C_2 + T(4) = \underline{C_2} + \underline{C_2} + \underline{C_2} + C_1$

. . . . .

$T(n) = C_2 * \log N + C_1 \implies T(n) = O(\log n)$

Example: Merge Sort



```
MERGE-SORT(A, p, r)
1  if $p < r$                                   C1
2       $q = \lfloor (p+r)/2 \rfloor$           C2
3       MERGE-SORT(A, p, q)      → 分成了一半
4       MERGE-SORT(A, q+1, r)
5       MERGE(A, p, q, r)                        C3
```

Call了别的函数所以要看那个函数

对于 Merge这个操作, C_3 可以被看作是 merge 中的一些基本操作, $C_3 n$是因为如果merge 拿到一个 input size为n, $C_3 n$ 是所需时间

$T(1) = C_1$ → Base case, 就是在最低点

$T(n) = C_2 + C_3 \times n + 2T(\frac{n}{2})$ → 相当关键

→ 在分析的时候, 记住 C_i 指的是在 i 行 执行一次 所需要的时间, n 指的是执行的次数

~~$T(1) = C_1$~~

$T(2) = C_2 + C_3 \times 2 + 2T(\frac{n}{2}) = C_2 + C_3 \times 2 + 2C_1$

$T(4) = C_2 + C_3 \times n + 2T(\frac{n}{2}) = C_2 + C_3 \times n + 2 \times (C_2 + C_3 \times 2 + 2C_1)$

$=$

...

$T(n) = (n-1) \times C_2 + n \times C_1 +$

应该放在abstract (with n) 一点的情况上走

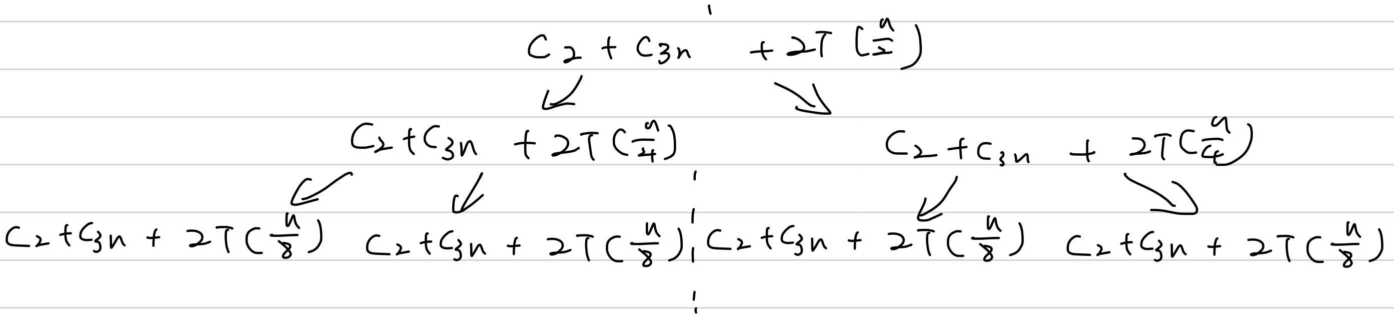
$$T(n) = C_2 + C_3 n + \boxed{2}\,T(\frac{\boxed{n}}{2})$$

$$= (1+2) \times C_2 + C_3(n+n) + \overset{\times 2}{\boxed{4}}\,T(\frac{\overset{\times \frac{1}{2}}{\boxed{n}}}{4})$$

$$= \underbrace{(1+2+4)}\,C_2 + C_3(n+n+n) + 8T(\frac{n}{8})$$

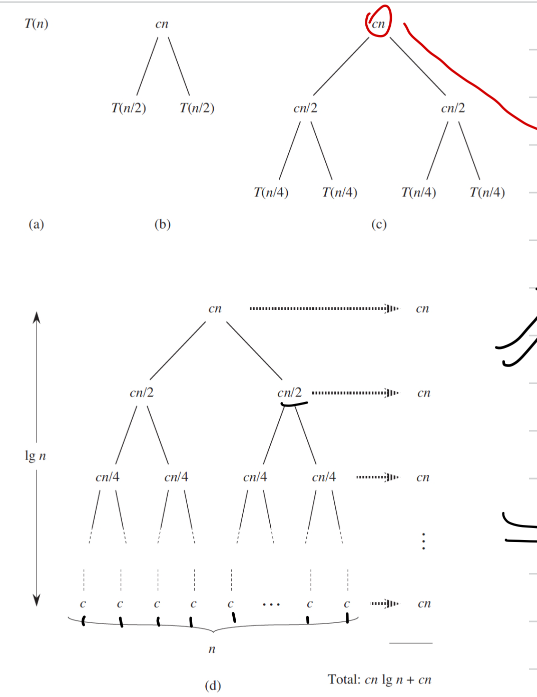
$$= (n-1)\,C_2 + \underbrace{C_3\, n\log_2 n} + n\,T(1) \Rightarrow T(n) = \Theta(n\log n)$$

→ $\log_2 N$ 个 $C_3 n$



$C_2 + C_3 n + 2T(\frac{n}{2})$
$C_2 + C_3 n + 2T(\frac{n}{4})$
$C_2 + C_3 n + 2T(\frac{n}{4})$
$C_2 + C_3 n + 2T(\frac{n}{8})$
$C_2 + C_3 n + 2T(\frac{n}{8})$
$C_2 + C_3 n + 2T(\frac{n}{8})$
$C_2 + C_3 n + 2T(\frac{n}{8})$

Reversion tree



Total: $cn \lg n + cn$

$T(1) = c$

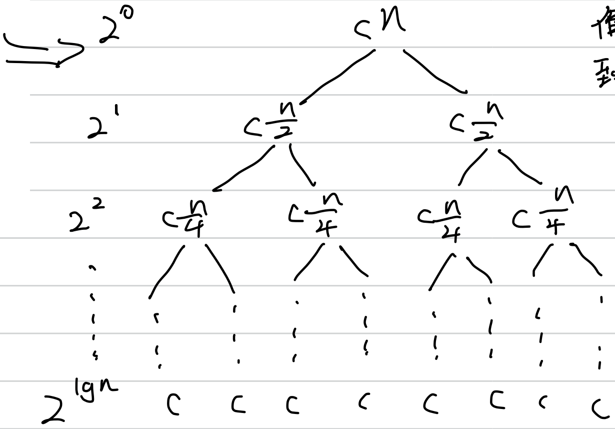
$T(n) = 2T(\frac{n}{2}) + cn$ for $n > 1$

cn 可以看作是在每一个 merge sort 运行时的 merge 的时间

每一行的cn代表了每次不同的input的 merge 跑的时间

→ $cn * \lg n$

最底端是 Base case 也就是当n=1的时候，根据 merge sort 的特性，在 Base case 中，也就不会再 recursively 执行 merge了



假设 c 代表着每次sort的 cost

到最后 n个c

→ 4个 $c \times \frac{n}{4} = cn$

→ cn

每一层都是 cn，总共有多少层？

总共有 lg n 层，但是因为

对于第一层来说，$\lg 1 = 0$

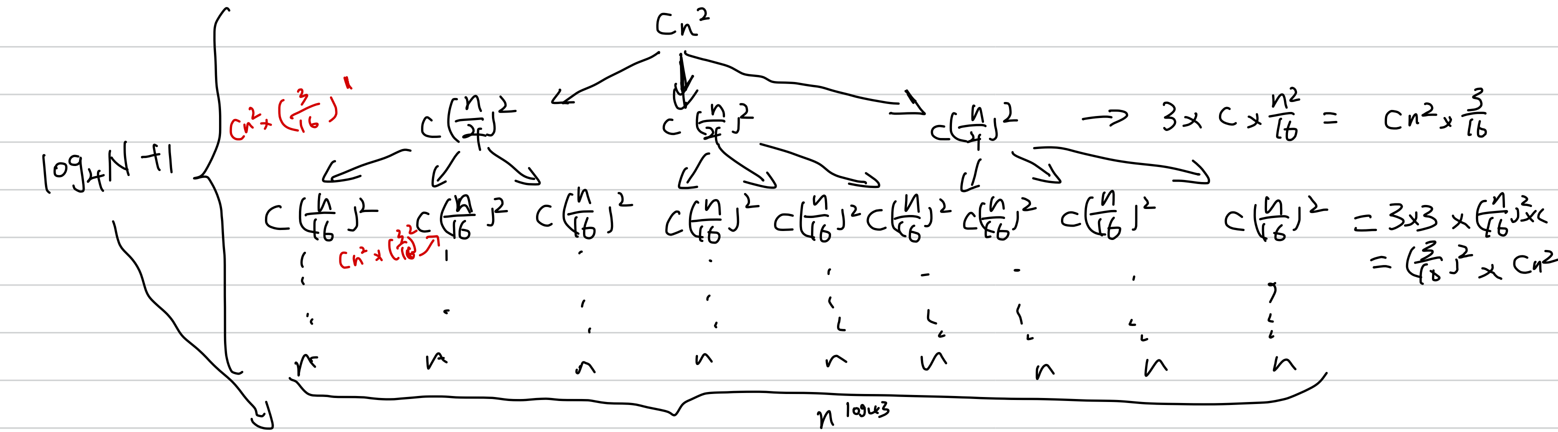
所以总共有 $\lg n + 1$ 层

$\therefore T(n) = cn \times (\lg n + 1) = cn\lg n + cn \in \Theta(cn\lg n)$

→在 recursion-tree 中： Each node represent the cost of a single subproblem, we sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per level cost to determine the total cost

A recursion tree example:

$$T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$$


Cn^2
$c(\frac{n}{4})^2$
$c(\frac{n}{4})^2$
$c(\frac{n}{4})^2$
$\rightarrow 3 \times C \times \frac{n^2}{16} = Cn^2 \times \frac{3}{16}$
$Cn^2 \times (\frac{3}{16})^1$
$c(\frac{n}{16})^2$
$Cn^2 \times (\frac{3}{16})^2$
$= 3 \times 3 \times (\frac{n}{16})^2 \times c$
$= (\frac{3}{16})^2 \times Cn^2$
$\log_4 N + 1$
n
$n^{\log_4 3}$

Level的递增 是 Subproblem每次下降多少来决定的

每一个level的数量是根据分成多少个Subproblem来决定的

↳ 在这里，每下降一层，就 $3 \times \underline{3^i}$ 个，

→ 上一层

所以在最底层 数量 $= 3^{\log_4 N} = n^{\log_4 3}$ 个

Total: $T(n) = Cn^2 + Cn^2 \times \frac{3}{16} + Cn^2 \times (\frac{3}{16})^2 + Cn^2 + (\frac{3}{16})^3 cn^2 + \cdots (\frac{3}{16})^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$

$$= \underbrace{\sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i}_{\downarrow} cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

$$= O(n^2)$$

Every time divide a problem into $\overset{3}{x}$ 个 subproblem of input $\overset{4}{y}$

层数: $L = \log_x N + \underset{\sim}{1}$ → (当 $N=1$ 的时候，($\log_x N = 0$，所以从0开始)

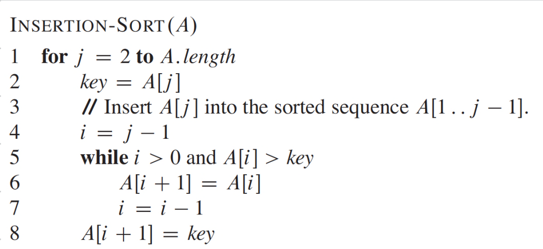
每一层个数: $g = y^{L-1} = g^{\log_x N}$

→ 2层: $4^{\log_3 N}$

Exercise

Lecture 02 :

* Rewrite the insertion - sort algorithm to sort into non-increasing instead of non - decreasing order



```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 . . j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

=>

```
for j=2 to A. length
    key = A[j]
    i=j-1
    while i>0 and A[i] < key
        A [i+1] =A [i]
        i= i-1
    A[i+1] = key
```

* What is the step cout of the insertion sort algorithm on the array $A=\{30, 41, 55, 23, 41, 52\}$

$6+5+4+3+2$

$\Downarrow$

20 steps

23 , 30 , 41 , 55 , 41 , 52

23 , 30 , 41 , 41 , 55 , 52

23 , 30 , 41 , 41 , 52 , 55

Lecture 03

3.1-1. Let $f(n)$ and $g(n)$ be asymptotically non negative functions. Using the basic definition of Θ notation, prove that $\max(f(n), g(n)) = \Theta(f(n)+g(n))$

$$\max(f(n), g(n)) \le 2 \times (f(n)+g(n))$$

$$\max(f(n), g(n)) \le f(n) + g(n)$$

$$2\max(f(n), g(n)) \ge f(n) + g(n)$$

$$\max(f(n), g(n)) \ge \frac{1}{2}(f(n)+g(n))$$

$$\therefore \frac{1}{2}(f(n)+g(n)) \le \max(f(n), g(n)) \le 2(f(n)+g(n))$$

$$\downarrow$$

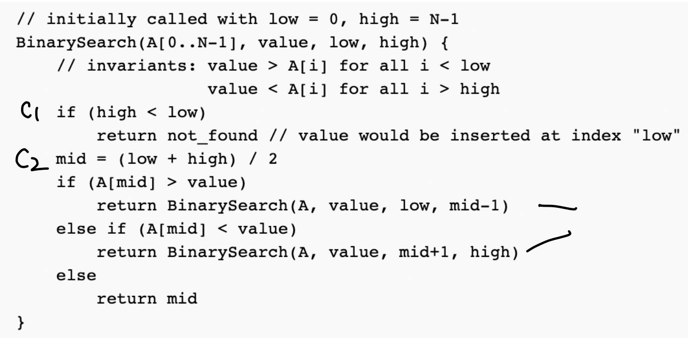
$$\therefore \max(f(n), g(n)) = \Theta(f(n)+g(n))$$

3.1-3. Explain why the statement, "The running time of algorithm A is at least $O(n^2)$." is meaningless

↳ Big-O notation implies an upper bound of a function, the statement however gives a lower bound on the running time of algorithm A, and it could mean there exist another upper bound for running time of algorithm. Therefore the statement contradicts with itself.

Lecture 04 :

What is the complexity of the recursive binary search algorithm?

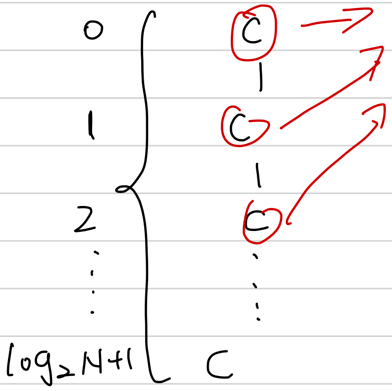


```
// initially called with low = 0, high = N-1
BinarySearch(A[0..N-1], value, low, high) {
    // invariants: value > A[i] for all i < low
                   value < A[i] for all i > high
C1  if (high < low)
        return not_found // value would be inserted at index "low"
C2  mid = (low + high) / 2
    if (A[mid] > value)
        return BinarySearch(A, value, low, mid-1)
    else if (A[mid] < value)
        return BinarySearch(A, value, mid+1, high)
    else
        return mid
}
```

$$\begin{cases} T(1) = C_1 & \text{base case} \\ T(n) = C_2 + T(\frac{n}{2}) \end{cases}$$

$$T(n) = C_2 + T(\tfrac{n}{2})$$
$$= C_2 + C_2 + T(\tfrac{n}{4})$$
$$= C_2 + C_2 + C_2 + T(\tfrac{n}{8})$$
$$\ldots$$
$$T(n) = C_2 * \log_2 N + C_1 = T(n) = \Theta(\log_2 N)$$

Recursion Tree analysis :



每一次都只需要进行简单的 c (get mid) 即可

$$\Rightarrow \quad C \times (\log_2 N + 1) = C \log_2 N + C$$

Lectur 05 :

Derive the complexity of the four different recurrence relations

↳ 指的就是前面的 A(n), B(n), Binary Search 和 Merge sort

```
def Q 1.3 (n):
        if n<=1 :                               1          Cn
                print n          $c_1$                      |
        else if n is odd:                       2          Cn
                print n          $c_2$                      |
                Q 1.3(n+1)                      3          $\frac{1}{2}$ Cn
        else :                                              |
                print n          $c_2$          4
                Q 1.3 ($\frac{n}{2}$)
```

$T(1) = C_1$

$T(2) = C_2 + T(\frac{n}{2}) = C_2 + C_1$

$T(3) = C_2 + T(n+1) = C_2 + T(4) \quad = \quad C_2 + C_2 + C_2 + 1$

$T(4) = C_2 + T(\frac{n}{2}) = C_2 + C_2 + 1$

$T(n) = C_2 + C_2 \times \log_2(n+1) + C_1$

NP- Completeness

* Polynomial-time algorithms : on input of size n, their worse case running time is $O(n^k)$ for some constant k

何为 verifiable?

if we were somehow given a "cretificate" of a solution, then we could verify that the certificate in time polynomial in the size of the input to the problem.

Decision Problem : A problem x is a decision problem if

- The answer to any instance of x is either Yes or No
- The answer to any instance of x is completely determined by the details of the instance

3种类的问题

- P : Decision Problems that are solvable in polynomial time
- NP : Decision Problems that are "verifiable" in Polynomial time
- NPC : 定义在后面

Non-Deterministic Turing machine : 是一种想象中的机器，这种机器你给它一个问题，它马上可以得到正确答案并验证，这是 NP的由来

↳

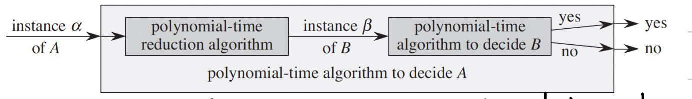
$P \subseteq NP$

可以看出，P / NP等对问题的难度进行分级的 technique 都仅仅是针对 Decision problem的，但可以把 Optimization problem → decision problem

↳ Shortest Path → Is a path with at most k edges exist?

到底 P == NP ? 如果我们可以找到 NP中最难的问题并且证明解决它只需 polynomial time，那么就可以证 P= NP. 怎么度量难度?

↳ Reduction ! "在解决一个问题之后，这个问题的答案可以解决另一个问题，那么另一个问题必定比直接解决的问题要简单"


instance α of A
polynomial-time reduction algorithm
instance β of B
polynomial-time algorithm to decide B
yes
no
yes
no
polynomial-time algorithm to decide A

Any instance of A can be transformed in polynomial time into an instance of B in an answer perserving way

P_1 : Given a set S of n integers, does S contain the value 4?

P_2 : Given a set S of n integers, does S contain the target integer k?

↳ P_1 比 P_2 简单 ⟹ P_1 reduce to P_2 ✓ ⟹ P_1 比 P_2 简单

P_2 比 P_1 简单 ⟹ P_2 reduce to P_1 ⟹ ✓

↓

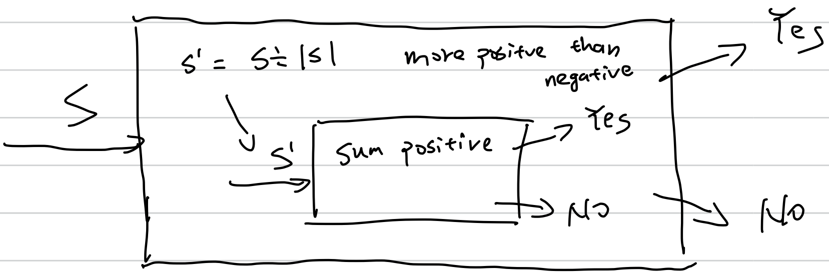
S' 变成 $S-k+4$ → 存在4即存在 k

Exercise:

A: Given a set S of n integers, are there more positive than negative integers in the set?

B: Given a set S of n integers, is the sum of the set positive?

Reduce problem A to problem B



SAT (Boolean satisfiability problem) 布尔可满足性问题

An instance of SAT is a boolean formula ϕ composed of

1、n boolean variables: $x_1, x_2, \dots, x_n$

2、m boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implication), $\leftrightarrow$ (iff

3、parentheses (括号)

↳ 问题是对于这个布尔表达式 E is there a way to assign True and False to the variables in E so that E is true / 换句话说, is E satisfiable?

↳ 并不是说 E 出来的是不是 true, 而是这个 E 中的 variable 中的设计能不能将它变成 true

Cook-Levin Theorem:

Let X be any problem in NP, then X can be reduce to SAT

↳ 世界上最难的问题 NP Complete: Every problem in NP can be reduced to NP-complete

↳ 所以如果可以证明一个 NP-complete 问题可以用 polynomial time 去解决, 相当于证明了所有 NP 中的问题都可以用 polynomial time 解决: P = NP

* A problem X is NP-complete iff:

$$\begin{cases} X \in NP \\ \forall Y \in NP,\ Y \text{ reduces to } X \end{cases}$$

或

$$\begin{cases} X \text{ in } NP \\ \exists Y \in NPC \text{ and } Y \text{ reduces to } X \end{cases}$$

证明方法

Example proof:

3- CNF is NP complete

→ CNF : expressed as an AND of clauses, each of which is the OR of one or more literals

→ 有3个

→ $\underbrace{(\underline{x_1} \vee \underline{\neg x_1} \vee \neg \underline{x_2})}_{3个} \wedge \underline{(x_3 \vee x_2 \vee x_4)} \wedge \underline{(\neg x_1 \vee \neg x_3 \vee \neg x_4)}$

3个

证明3-CNF is NP Complete → 证 SAT reduce t 3CNF-SAT

*CNF - SAT 是一个 NP Complete

Example problem : K- clique problem : Given a graph G and an integer K, does G contain a set of K vertices that are all directly connected by edges?

⇒ Reduce CNF-SAT to K- clique

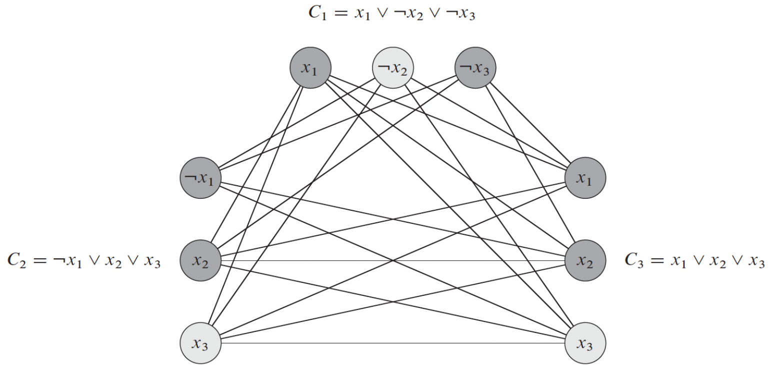
CNF-SAT:

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

↳ 通过给 variable boolean 值，可否另 E 为 true

↳ 每个 clauses 出一个 vertex, $a \wedge b \wedge c$ 为 true iff 在一个设计好的 graph 中存在 $a \to b \to c$ 的 connected graph

G →



$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$
x_1
$\neg x_2$
$\neg x_3$
$\neg x_1$
x_2
x_3
$C_2 = \neg x_1 \vee x_2 \vee x_3$
x_1
x_2
x_3
$C_3 = x_1 \vee x_2 \vee x_3$

所以如果存在一条 K- clique 的话，则必定有一种布尔变量的组合可以另 E 为 true

* Structure of NP- completness

CIRCUIT - SAT

↓

SAT

↓

CNF - SAT

↙ CLIQUE

↘ SUBSET - SUM

CLIQUE

↓

VERTEX COVER

↓

HAM- CYCLE

↓

TSP

K-clique to K-cover

K-cover : Let G be a graph (V, E). A vertex cover of G is a subset S of V with the property that every edge in E has at least one end in S. A K-vertex cover that contains excatly K vertices.

G　　$\overline{G}$

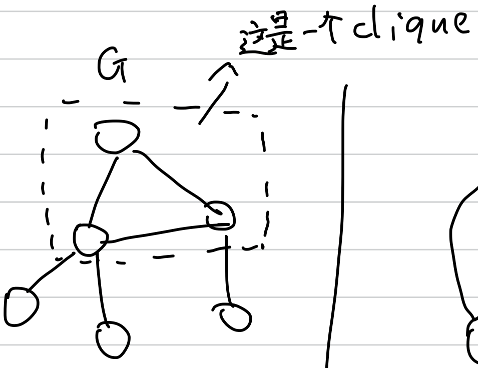
↘ ↙

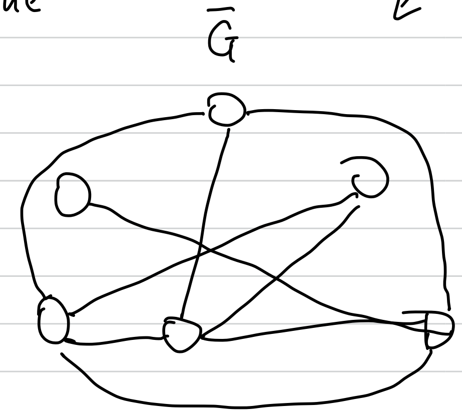
组成一个complete的 graph

↳如果在$\overline{G}$中存在 cover V (V是S的子集，cover V是一组vertex并且在一个graph中所有的edge都至少有一个end point在这个cover V的vertex中)

K是 K-clique 中的 K，在G中有 K个vertex可以组成connected graph

所以如果在 K-cover 问题中，在$\overline{G}$中存在 n-K个 vertex 组成的vertex cover,





如果这是一个 clique的话 → vertex 两两相连

↓

必不可能组成$\overline{G}$的vertex cover

如果在$\overline{G}$中，存在(n-K)个 vertice 组成的 vertex cover

⇓

这K个没有组成 vertex cover的 vertices 之间并不存在任何一条edge，不然它们就会在 vertex cover 中

⇓

因为G中存在 所有 $\overline{G}$ 中没有的 edge

⇓

所以在G中这 K个 vertices 中间两两相连

⇓

K-clique!

Textbook 一些补充的知识：

* Euler tour: An Euler tour of a connected, directed graph $G=(V, E)$ is a cycle that traverse each edge of G excatly once, although it's allowed to visit each vertex more than once.

*Hamiltonian cycle: A hamiltonian cycle of a directed graph $G=(V, E)$ is a simple cycle that contains each vertex in V.

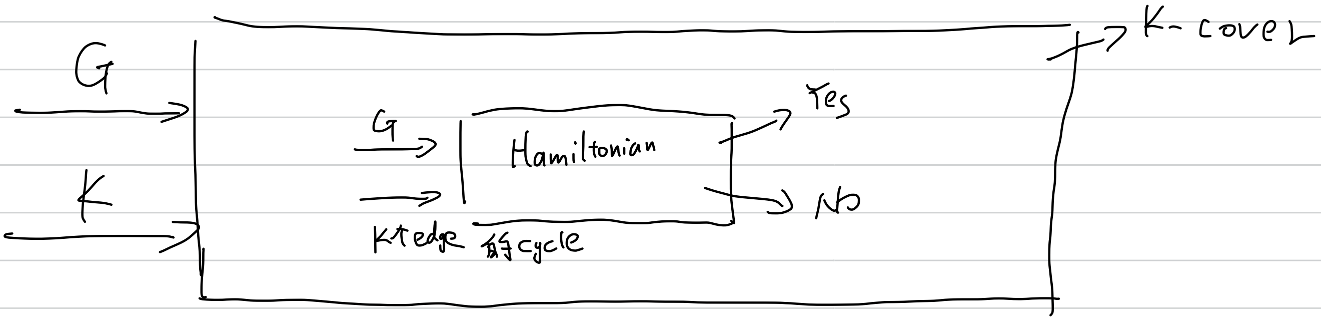
Show that Hamiltonian Cycle problem is NPC

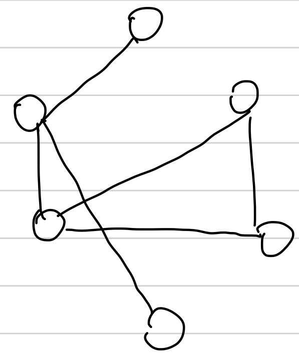
已经知道是 NPC : $\begin{cases} \text{SAT} \\ \text{K vertex Cover} \\ \text{K clique} \end{cases}$

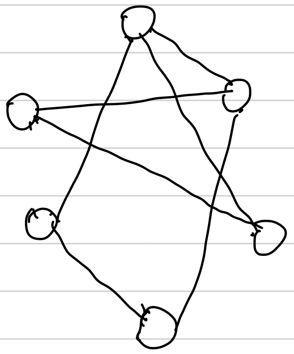
SAT reduce to K clique reduce to K vertex

*Hamiltonian cycle: A hamiltonian cycle of a directed graph G=(V,E) is a simple cycle that contains each vertex in V.

↳ Given a graph G and an integer K, does G







V

e'

set of edges

↓

edge = e

↓

V

↓

存在 hamilto cycle

↓

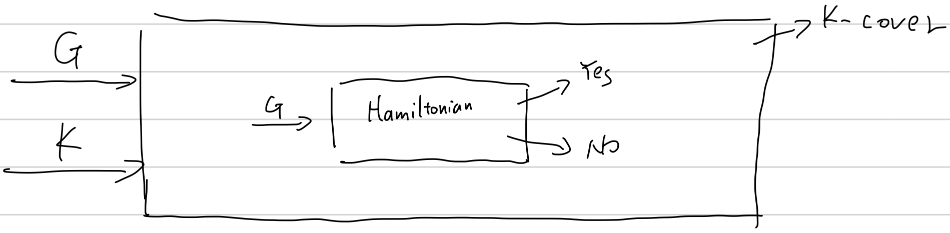
存在一个集合中有K个edge 的cycle

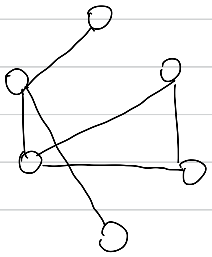
$$e + e' = 1+2+3+\cdots+V$$

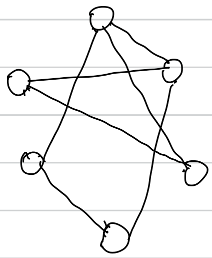
$$= \frac{(V+1)\times V}{2}$$

*Hamiltonian cycle: A hamiltonian cycle of a directed graph $G=(V,E)$ is a simple cycle that contains each vertex in V.

↳ Given a graph G and an integer K, does G







v

e'

set of edges

√

K-cover: 存在 K个 vertice 从而另所有的 edge 都至少有一头在 K个 vertice 中

↓

存在 hamilto cycle

↓

存在一个集合中有 K 个 edge 的 cycle

$$\hookrightarrow \underbrace{\frac{(v+1)\times v}{2}} - K$$

$$\text{edge} = \overset{v}{\underset{v}{e}}$$

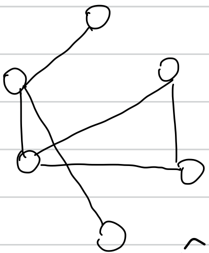
$$e + e' = 1+2+3+ \cdots + v$$

$$= \frac{(v+1)\times v}{2}$$

至多 $v(\frac{1}{2}v-\frac{1}{2})$ 条 edge

存不存在 K个 vertex
such that every edge ($v(\frac{1}{2}v-\frac{1}{2})$) 中至少有一个 end在里面

3



$K <= v(\frac{1}{2}v-\frac{1}{2})$

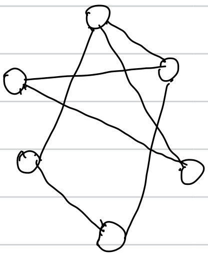
$v(\frac{1}{2}v-\frac{1}{2}) \geq K$

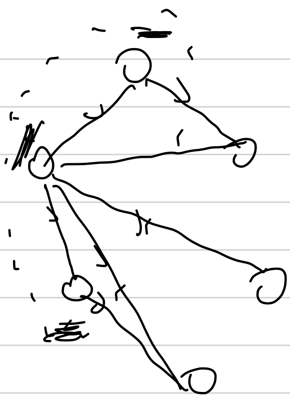
$$\boxed{\frac{v(\frac{1}{2}v-\frac{1}{2})}{v}}$$

个 →

K-cover: 存在 K个 vertice 从
而另所有的 edge 都至少有一头
在 K个 vertice 中

⇓ K





↓

存在 hamilto cycle

↓

在G中 存在至少 V条 edge

总edge 数量: $\frac{1}{2}v(v+1)$

$\frac{1}{2}v(v-1)$

至多 $\frac{1}{2}v(v+1) - v$

$\frac{1}{2}v^2 + \frac{1}{2}v - v$

$\frac{1}{2}v^2 - \frac{1}{2}v$

至多 $v(\frac{1}{2}v-\frac{1}{2})$ 条 edge

↓

在G中, K- vertex cover
中存在 K个 vertices such that
跟 $v(\frac{1}{2}v-\frac{1}{2})$ 条edge中的每一条相连

⇒

$K <= V$

$v(\frac{1}{2}v-\frac{1}{2})$ 条 edge

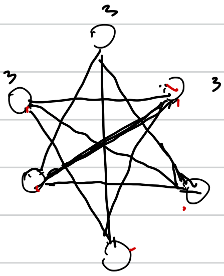
$6 \times (\frac{1}{2} \times 6 - \frac{1}{2})$

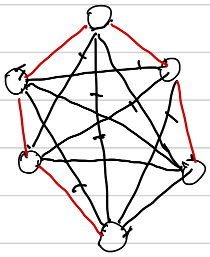
$6 \times 3 - \frac{1}{2}$

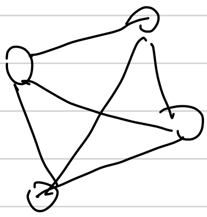
假设存在 K- clique : G中存在 至少

→ 15条

at most 6 edge



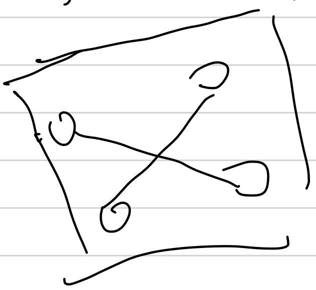




6

3+2+1

4



6+5+4+3+2 +1

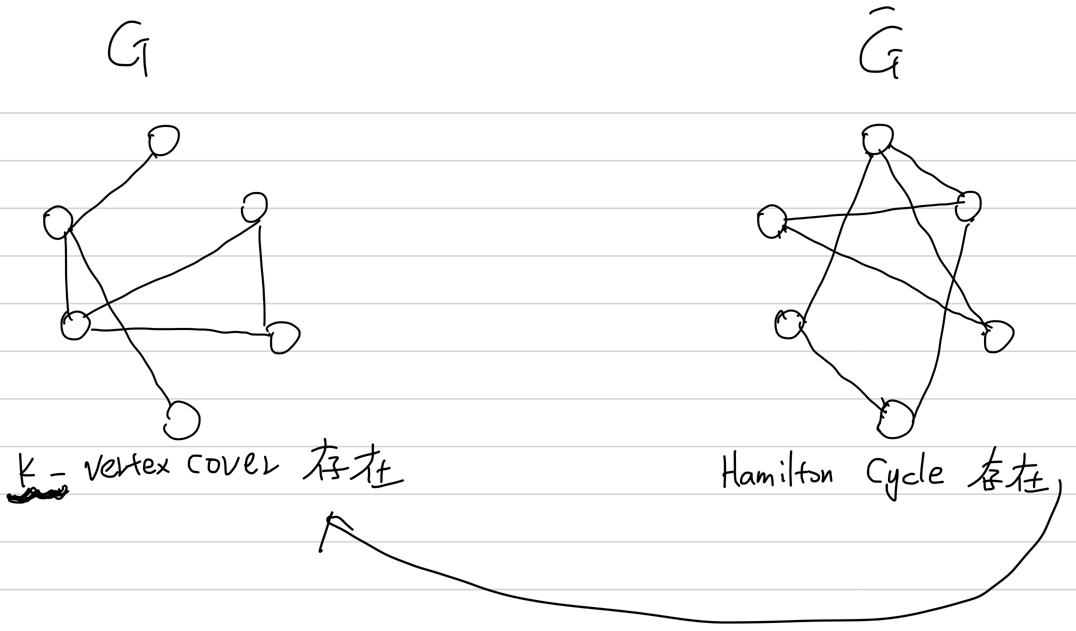
总: $v + (v-1) + (v-2) + \cdots + 2 + 1$

$\frac{1}{2} \times v \times (v+1)$

$\frac{1}{2} \times v \times (v+1) - v$

$\frac{1}{2}v^2 + \frac{1}{2}v - v$

$\frac{1}{2}v^2 - \frac{1}{2}$



如果(假设) Hamilton Cycle 存在：
在 $\bar{G}$ 中至少 存在 V条 edge

G 中 K- vertex cover是否存在?
在G中至多存在 $\frac{1}{2}V(V+1) - V$ 条edge

Divide and Conquer

↳ Divide and Conquer 是一种算法的类别，也是一种解决问题的方法

↳ formal definition: In divide and conquer, we solve a problem recursively, applying three steps at each level of the recursion:

- Divide the problem into a number of subproblems that are smaller instances of the same problem
- Conquer the subproblems by solving them recursively.[当然如果是落在了 base case 中的话，直接解决这个问题]
- Combine the solutions to the subproblems into the solution for the original problem

Complexity for divide and conquer:

※ Suppose that the division of the problem yields a sub problems, each of which is $\frac{1}{b}$ the size of the original (It takes time $T(\frac{n}{b})$ to solve one Sub-problem of size $\frac{n}{b}$), $D(n)$ time to divide the problem into sub problems, $C(n)$ to combine the solutions to the subproblems.

$$T(n)=\begin{cases}\Theta(1) & \text{Base case}\\ \underbrace{aT(\frac{n}{b})}_{a\text{个subproblem of size }\frac{n}{b}} + \underbrace{D(n)}_{\text{divide}} + \underbrace{C(n)}_{\text{combine}} & \text{otherwise}\end{cases}$$

※ A recursive algorithm might divide subproblem in different sizes

* 而且 subproblem 的 size 也不一定是 fraction of the original problem size

Example D&C Algorithm:
- Binary Search
- Quicksort
- Merge sort
- ....

Binary Search:

这个值是自己定义的，似乎只要是一个常数就可以了

```
Binary_Search (A, t) :                # Assume A is already sorted
    if A has less than 4 elements :
        perform sequential search 直接硬找
    else:
        Divide A into two equal subsequences
        Compare t with the last element of the first half of A (mid)
        if t < mid :
            Binary Search (first half)
        else :
            Binary Search (second half)
```

$$\begin{cases}T(n)=\Theta 1 & n\le 4 \rightarrow \text{Base Case}\\ T(n)=T(\frac{n}{2})+\Theta(1), & n>4\end{cases}$$

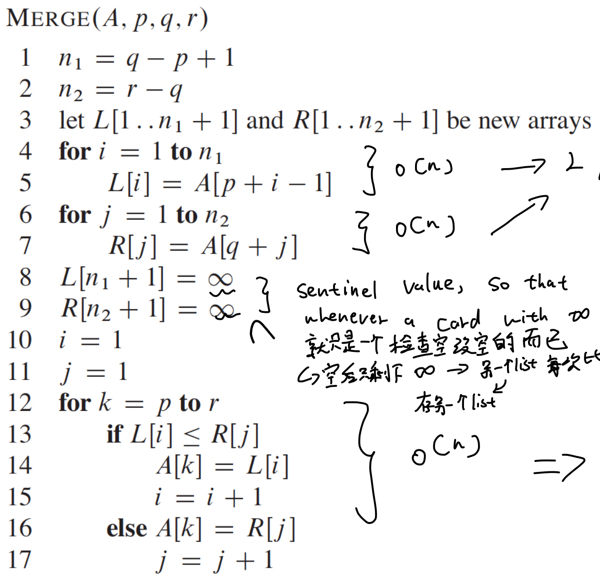
$\hookrightarrow T(n)=O(\log n)$

$$\Rightarrow T(n)=\begin{cases}T(1)=\Theta(1), & n\le 4\\ T(n)=\underbrace{1\times T(\frac{n}{2})}_{\text{只会分出 1个 subproblem, 因为只会找一个 half}}+\underbrace{D(n)}_{1}+\underbrace{C(n)}_{1}\end{cases}$$

Merge Sort: Divide the n-element sequence to be sorted into two subsequences of $\frac{n}{2}$ elements each, sort the two subsequences recursively using merge sort, then merge the two sorted arrays.

↳ 最关键的操作是 merge (A, p, q, r) => A是一个 array
A[p ⋯ q] 是一个 sorted array
A [q+1, r] 是另一个 sorted array
⇓ return
Sorted Array A (p − r)

↓ 难的是 combine



```
MERGE(A, p, q, r)
n_1 = q - p + 1
n_2 = r - q
let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
for i = 1 to n_1
    L[i] = A[p + i - 1]
for j = 1 to n_2
    R[j] = A[q + j]
L[n_1 + 1] = ∞
R[n_2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

} O(n) → L/R 中存入原 A 前半/后半的内容

} O(n) ↗

} sentinel value, so that whenever a card with ∞ is exposed
就是一个检查空没空的而已
(空后就补 ∞ → 另一个list 每次比对
存另一个list

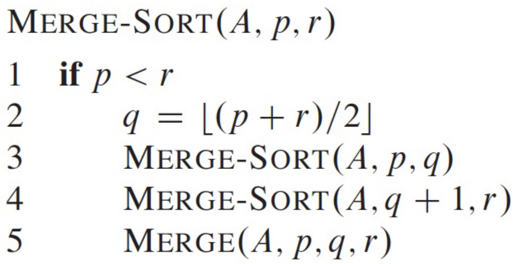
} O(n) => L[ ] R[ ] A[ ]

=> Merge 的 running time $T(n) \le O(n)$

一个一个对比, min(L[i], R[j])

这两个都 sort 好的前提下

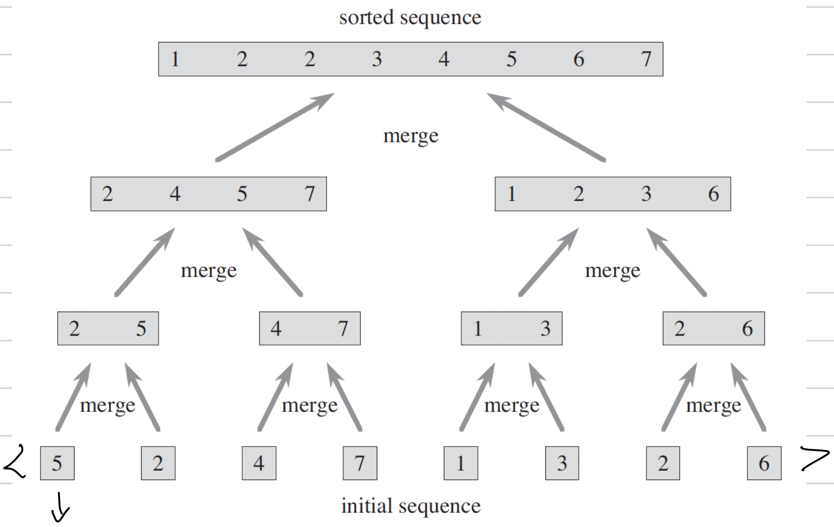
Merge Sort 的 Pseudo code



```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Analyzing divide and conquer algorithm

* 含有 recursive call 的 algorithm 的 running time 被称为 recurrence equation or recurrence = running time on a problem of size n



base case

对于 Merge Sort:

$$\begin{cases} T(n) = \Theta(1) & n \le 4 \\ T(n) = 2 \times T(\frac{n}{2}) + \Theta(n) & , n > 4 \end{cases}$$

→ Base Case

↳ $T(n) = \Theta(n\log n)$

$$\begin{cases} T(1) = \Theta(1) \\ T(n) = 2 \times T(\frac{n}{2}) + \underbrace{C(n)}_{\downarrow \, n} + \underbrace{\Theta(n)}_{\downarrow \, 1} \end{cases}$$

merge 的时间

Robin Dawes' :

```
Merge Sort (A):
    if A has <= $\boxed{5}$ elements:
        Sort A using any method
    else:
        MergeSort (left half of A)
        Merge Sort (right half of A)
        Merge the two sorted half together into one full sorted set
```

这只是它自己试出来的，这里可以是 any constant

↳ $T(n) = C_1$ → Base Case : $n <= 5$

$T(n) = C_2 + C_3 * n + 2T(\frac{n}{2})$ $n > 5$

如果我们想改进一下 merge sort，令 merge sort 每次 divide into three parts?

↳ 没什么用，因为虽然 recursive call 变少了，但是因为每次要 merge 三个 array，所以这个时间也变长了

$T(n) : C_2 + C_3 * n + 3 * T(\frac{n}{3}) \implies \Theta(n\log n)$ 还是一样的

Quick sort :

```
Quick-Sort (A):
    if A has less than $\boxed{4}$ elements:
        Sort A using any algorithm 硬来
    else:
        $q$ = partion (A)
        Quick-Sort (A[1, ..., q-1])
        Quick-Sort (A[q, ..., n])
```

同样的，any constant

在A中 randomly 挑选一个 element q，把所有比q小的放在 q 的左边，所有比 q 大的放在q的右边

目标是找到一个可以正好处于 midpoint 的点..，

Worst Case : 每次选中的 q 都在处理后变成倒数第二位，从而使得下一个 quick sort 要 sort n-1

如果每次都只找到 ↓ 这个位置，相当于有 n 个 recursive call

$$\begin{cases} T(n) = \Theta(1) & , n \le 4 \\ T(n) = T(n-1) + \Theta(n) & , n > 4 \end{cases}$$

↳ $T(n) = \Theta(n^2)$

只有 1个 n-1的，因为剩那个只有一个数，

Best case : 每次选中的q都在处理后变成 midpoint，这样每次都正好把所有 element 分至 2半

$$\begin{cases} T(n) = \Theta(1) , & n \le 4 \\ T(n) = 2T(\frac{n}{2}) + \Theta(n) & , n > 4 \end{cases}$$

↳ $T(n) = \Theta(n\log n)$

Maximum Subarray Problem

The maximum - subarray problem :

* Given an Array A of n integers, can we find a subarray A[i,⋯⋯,j] , where the sum of its elements is the maximum?

↳ Brute force : $\Theta(n^2)$, 遍历全部的可能

Special case :

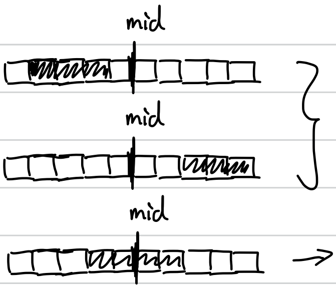
A 中全部 value 都 < 0 : 直接找最大

A 中全部 value 都 > 0 : 直接全加起来

```
Maximum_Subarray (A) :                                   T(n) in Θ(logn)
    if |A| ≤ 5 :
        Find the maximum - subarray using brute force
    else :
        Divide the array into A[1, ..., mid] and A[mid+1, ..., n]
        Maximum_Subarray (A[1,...., mid])
        Maximum_Subarray (A[mid+1, ...., n])
        Return one of the subarray with a greater sum
```

只能找出 |Subset| ≤ 5 的 Subset Sum 的最大

给定一个 array, 这个 subarray 可能出现在

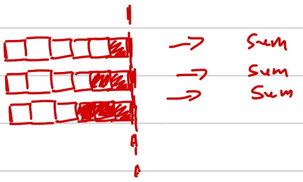


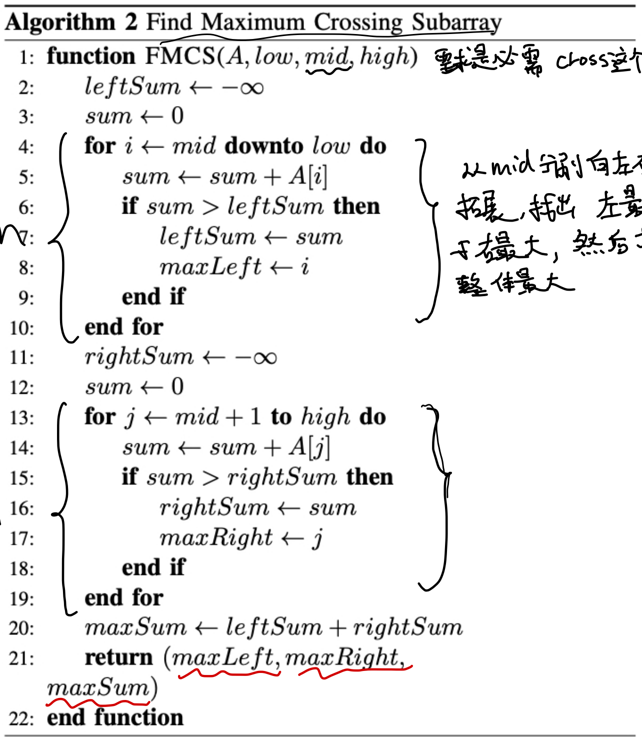
前面的算法已经覆盖到了

因为目前我们 assume 的算法是找两半, 如果最大的 Subset 实际存在于中线的位置

→ what if the subarray happen here?

Max - Crossing - Subarray (A)





Algorithm 2 Find Maximum Crossing Subarray

```
function FMCS($A, low, mid, high$)
    $leftSum \leftarrow -\infty$
    $sum \leftarrow 0$
    for $i \leftarrow mid$ downto $low$ do
        $sum \leftarrow sum + A[i]$
        if $sum > leftSum$ then
            $leftSum \leftarrow sum$
            $maxLeft \leftarrow i$
        end if
    end for
    $rightSum \leftarrow -\infty$
    $sum \leftarrow 0$
    for $j \leftarrow mid + 1$ to $high$ do
        $sum \leftarrow sum + A[j]$
        if $sum > rightSum$ then
            $rightSum \leftarrow sum$
            $maxRight \leftarrow j$
        end if
    end for
    $maxSum \leftarrow leftSum + rightSum$
    return ($maxLeft, maxRight, maxSum$)
end function
```

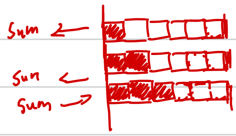
要找是必需 cross这个 mid

n

从 mid 分别向左右拓展, 找出左最大 右最大, 然后加在一起 整体最大

n

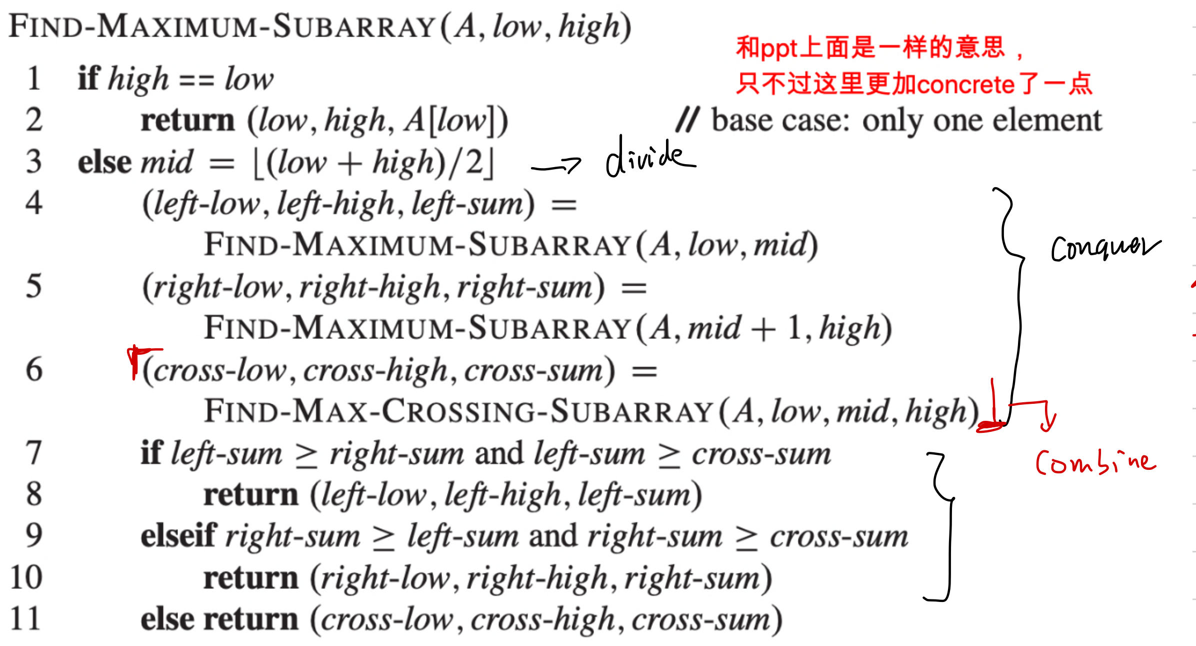
⇒ T(n) is in O(n)



```
Maximum_Subarray (A) :
    if |A| ≤ 5 :
        Find the maximum_subarray using brute force
    else :
        Divide the array into A[1, ..., mid] and A[mid+1, ..., n]
        Maximum_Subarray (A[1, ..., mid])  → 找到左半边最大的
        Maximum_Subarray (A[mid+1, ..., n])  → 找到右半边最大的
        Max-crossing-subarray (A)  → 找到穿过中线的最大的
        Return one of the subarray with a greater sum (三个里面找一个最大的)
```

$$\begin{cases} T(n) = C_1, & n \le 5 \\ T(n) = 2\times T(\frac{n}{2}) + \overset{\text{Max-crossing}}{C_2 \times n}, & n > 5 \end{cases}$$

$\rightarrow T(n)$ is in $\Theta(n\log n)$



```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋          → divide
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Conquer

Combine

和ppt上面是一样的意思，
只不过这里更加concrete了一点

第六行其不属于conquer是因为，FAMS并没有操作一个smaller part of the original problem size

Finding the closet pair of points

Problem: Finding the closet pair of points in a set P in metric space of $n \geq 2$ points

* Distance is measure in Euclidean distance

↳ 在二维平面内的n个点中，找出欧式距离最近的两个点。

Brute force: 万物皆可穷举解决：

```
Min = ∞ ;
For every point i in the set:
    For every other point j:
        calculate the distance between (i, j)
        if distance (i, j) < = ∞ :
            Smallest = i, j
return smallest
```

$O(n^2)$

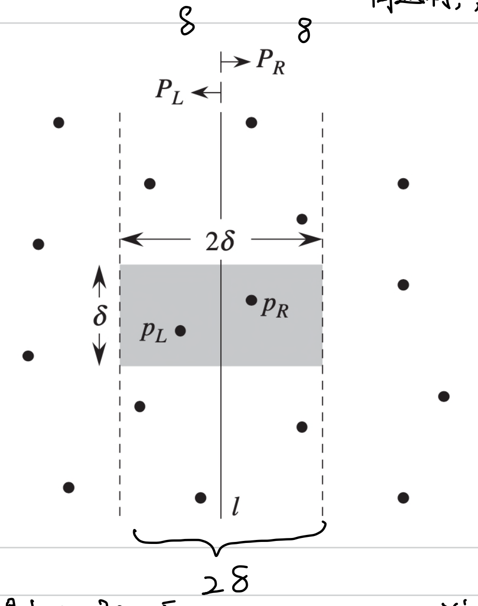
Divide and conquer:

Divide: 把整个 set P 从中间处用条线分开，两个 subproblems，P_L 和 P_R

Conquer: 分别找出，刚刚分出的 P_L 和 P_R 中的最小；令 $\delta = \min(P_L, P_R)$

Combine: 这时，全局最小的要么是 P_L 最小，要么是 P_R 最小，要么是(一个点在 P_L 中，一个点在 P_R 中构成)的最小

问题来了，怎么找到一个点在左，一个点在右的最小的？



a、首先，如果存在一对距离比 δ 还小的点，并且一个在左，一个在右，这对点必定存在于如图 2δ 中间的区域

↳ 为什么是 2δ，而不是 δ？ → 如果是 δ 的话，一边只有 $\frac{1}{2}\delta$，并不可以把全部的可能性给涵盖掉

b、(具体怎么找？)：先有一个 array Y'，Y' 中存了所有的在这个 vertical strip 中的点。

c、

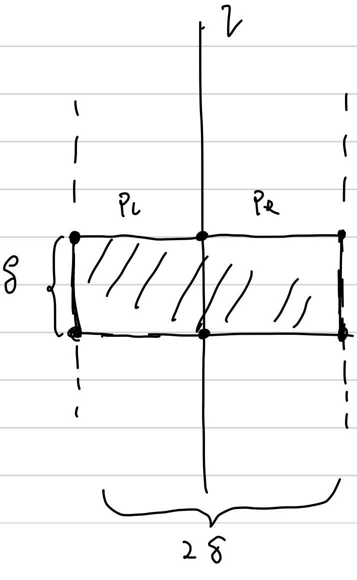
```
For each point in Y' (从下向上走)
    Try to find points in Y' that are within δ unit with current point
    if distance < current Small :
        current Small = current_pair
```

这一步依然需要

问题：即使把问题映射在了一个相对小的空间里面，即使只有 $\frac{1}{10}n$，但貌似 $\frac{1}{10}n \times (\frac{1}{10}n - 1) \approx n^2$, what difference?

课本证明了，For every point in Y'，只需检查 7 个点。

↘ 为什么？：



因为目前找到的左半区最小的距离 = δ

所以左半区在这个区域的点不超过 4 个，因为如果有其它的点存在于左半区，它们的距离将会小于 δ

同理，在右半区最多只有 4 个

因为中间线上的点，有可能会同时归到左和右，所以至多有 4+4=8 个点、

但 PPT 上写的是 6 个 (把中线的点合并)

∴ For every point in Y', at most 6 examination to run

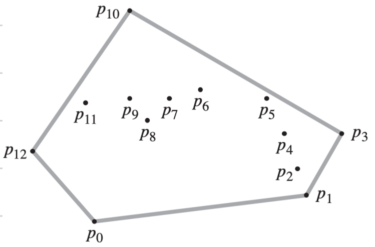
Pseudocode:

```
closet-pair(P):
if $|P| \leq 4$:
    Find the closest pair using brute-force
else:
    Divide the set into a left half $P_L$ and a right half $P_R$
    $\delta_L$ = Closet-Pair($P_L$)
    $\delta_R$ = closet-Pair($P_R$)
    $\delta$ = min($\delta_L$, $\delta_R$)
    $Y'$ = 全部位于离中线 $\delta$ 范围的点.
    遍历找出在 $Y'$ 中距离最小的 pair
    if $\delta' <= \delta$:
        return $Y'$ 中最小
    else:
        return $P_L$ / $P_R$ 中最小
```

$$T(n) = \begin{cases} T(1) = \Theta(1) \\ T(n) = 2 \times T(\frac{n}{2}) + \underbrace{Cn}_{\text{找 Panel 最小}} \end{cases}$$

Convex Hull

The convex hull of a set Q of points, is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior. (指的是最小的能容纳下所有点的凸多边形)



→ 所以这个问题就是希望可以在一个 given n points 中找到 Convex hull

Brute force: $O(n^3)$

Divide & Conquer Algorithm

→ 和之前一样...

Divide 先把这些点 sorted好，然后再 P_L, P_R (取中线分)

Conquer 在 P_L, P_R 中分别找到 convex hull

Combine 把 P_L, P_R 中的 convex hull 中 combine 起来

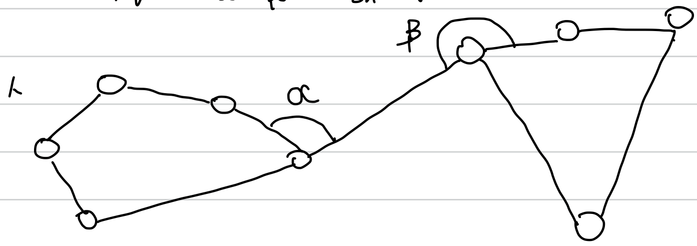
方便起见，可以把 base case 设置为 3，因为最小当点的数量为 3 时就可以组成图形了.

问题是，该怎么 merge (combine) ???

→ 如果每次都只是把左最高与右最高给连在一起，左最低与右最低相连，会造成部分的点没能被包括

解决办法：

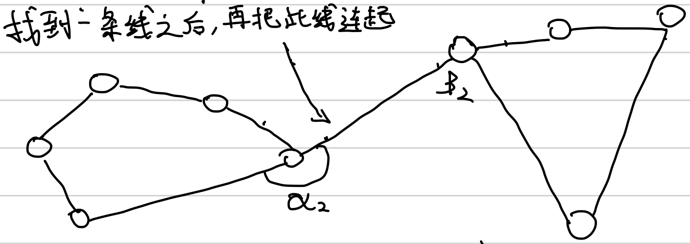
1、Connecting the right most point on left's convex hull to the left most point on the right



2、循环式地找，直到找出一条 edge令 α 和 β 都大于180°

→ 每次替换小于 180°的那边的点。

3、找到一条线之后，再把此线连起



4、同上，找出令 α_2，β_2 均大于 180°的线

O(n)

```
Convex-Hull ($Q$):
  if $|Q| \le 3$:
      Return all three nodes as the CH($Q$)
  else:
      Divide points in $Q$ into $Q_L$ and $Q_R$
      Convex-Hull($Q_L$)
      Convex-Hull($Q_R$)
      Merge the CH($Q_L$) and CH($Q_R$) into CH($Q$)
      Return CH($Q$)
```

$T(n) = c_1, n \le 3$

$T(n) = 2 \times T(\frac{n}{2}) + c_2 \times n, n > 3$

$O(n \log n)$

此处合并的complexity是o(n)

Jarvis's march algorithm

1、就像包礼物一样，从一个必在 convex hull 上的点出发，这里选取了最左边的点。

↳ complexity

O(n*h) → 因为对于每一个找到的点，都需要计算剩下的所有的点和它们的角度

↓

当 n中全部点都在 convex hull 中

O(n²) (worst)

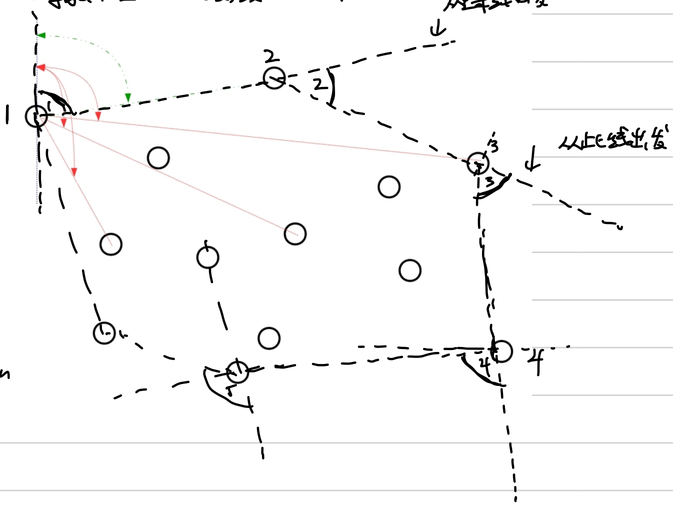
当 convex hull has no more than k 个 vertex：O(h*n)

↳ O(n)

1、找出和这一点（角度）最近的点

从这条线出发

从此线出发



Subset Sum

Given a set S of n integers and a target value k, does S have a subset that sum to k ?

Brute force：把全部 2^n 的可能性都走遍，看看有没有符合要求的 ⟹ $O(2^n)$

Pair Sum：

Given a set s of n integers and a target integer k, does S contain a pair of values that sum to k?

⟹ 先把 S 用升序给排起来

令 $t = s_1 + s_n$

如果 $t == k$：找到

如果 $t < k$：把 s_1 给删了，因为 s_1 + 最大的 s_n 都不行，加别的肯定也不行

如果 $t > k$：把 s_n 删了，同上

Complexity：$\underbrace{O(n * \log n)}_{\text{sorting}} + \underbrace{O(n)}_{\text{实际} \rightarrow \text{每次都减去 1个}} \rightarrow O(n\log n)$

⇓

Two set Pair-Sum：

Given sets X and Y with n elements in each set, and a target integer k, is there an $x \in X$ and a $y \in Y$ such that $x + y = k$?

↳ 基本一样，先 sort X 和 Y，(这里 assume 必定一个在 X，一个在 y 的情况下)

令 $t = x_1 + y_n$

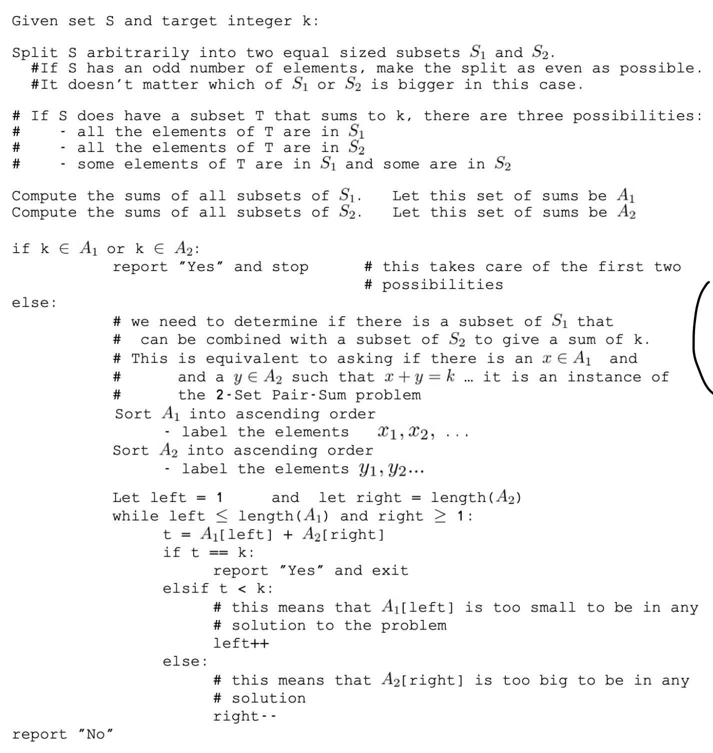
如果 $t == k$：找到

如果 $t < k$：把 x_1 给删了，因为 s_1 + 最大的 s_n 都不行，加别的肯定也不行

如果 $t > k$：把 y_n 删了，同上

Complexity：$\underbrace{O(n\log n)}_{\text{sort}} + \underbrace{O(n)}_{\text{寻找 pair 的过程}}$

So! 一开始很难搞定的 subset sum 问题，可以变成：



```
Given set S and target integer k:

Split S arbitrarily into two equal sized subsets $S_1$ and $S_2$.
  #If S has an odd number of elements, make the split as even as possible.
  #It doesn't matter which of $S_1$ or $S_2$ is bigger in this case.

# If S does have a subset T that sums to k, there are three possibilities:
#     - all the elements of T are in $S_1$
#     - all the elements of T are in $S_2$
#     - some elements of T are in $S_1$ and some are in $S_2$

Compute the sums of all subsets of $S_1$.   Let this set of sums be $A_1$
Compute the sums of all subsets of $S_2$.   Let this set of sums be $A_2$

if k ∈ $A_1$ or k ∈ $A_2$:
           report "Yes" and stop       # this takes care of the first two
                                       # possibilities
else:
           # we need to determine if there is a subset of $S_1$ that
           #  can be combined with a subset of $S_2$ to give a sum of k.
           # This is equivalent to asking if there is an $x \in A_1$  and
           #       and a $y \in A_2$ such that $x + y = k$ … it is an instance of
           #       the 2-Set Pair-Sum problem
           Sort $A_1$ into ascending order
                - label the elements   $x_1, x_2, \ldots$
           Sort $A_2$ into ascending order
                - label the elements $y_1, y_2\ldots$

           Let left = 1      and  let right = length($A_2$)
           while left ≤ length($A_1$) and right ≥ 1:
                t = $A_1$[left] + $A_2$[right]
                if t == k:
                     report "Yes" and exit
                elsif t < k:
                     # this means that $A_1$[left] is too small to be in any
                     # solution to the problem
                     left++
                else:
                     # this means that $A_2$[right] is too big to be in any
                     # solution
                     right--
report "No"
```

一开始的 subset sum → 有 n 个 element，所以共有 2^n 个 subset，一个一个地比对它们的大小，所以 complexity $= O(2^n)$

↘

现在把 S_1 和 S_2 中分别只有 $\frac{n}{2}$ ↗ elements，所以硬找变成了 $2 * O(2^{\frac{n}{2}})$

A_1 A_2 是 S_1 和 S_2 的所有 set → $|A_1| = |A_2| = 2^{\frac{n}{2}}$

↳ sort A_1 和 A_2 (现在把这个东西丢到 2 set pair sum 中)

⟹ Complexity of sort: $\underbrace{2^{\frac{n}{2}}}_{\text{merge}} * \log n * 2 \leq O(n * 2^{\frac{n}{2}})$

⟹ 然后是 2 set pair-sum → $2 * O(2^{\frac{n}{2}})$

⟹ $2 * O(2^{\frac{n}{2}}) + O(n * 2^{\frac{n}{2}}) + 2 * O(2^{\frac{n}{2}})$

⇓

$O(n * 2^{\frac{n}{2}})$

Exercise:

Finds three values from A, B, C, sums to a target K

(we can call the Two set Pair Sum function we wrote)

① 把 set A,B 中每一个 pair 都存入 set D 中, => O(n^2)

② set D 和 set C, target K 丢入 Two set Pair Sum 中

34.5-4: Show how to solve the subset-sum in polynomial time if the tagen value is expressed in unary

Unary: 5 : |||||

33.3-3 : Prove that the pair of points farthest from each other must be vertices of CH(Q)

↳ Assume there exist a convex hull which doesn't include the pair of points farthest from each other : for every edge that forms convex hull, it has length that doesn't longer than the farthest edge exist

↓

there exist at least one point that doesn't lay inside the convex hull

↓

it does not on the convex hull because that's what the assumption say

↓

this convex hull is not a convex hull

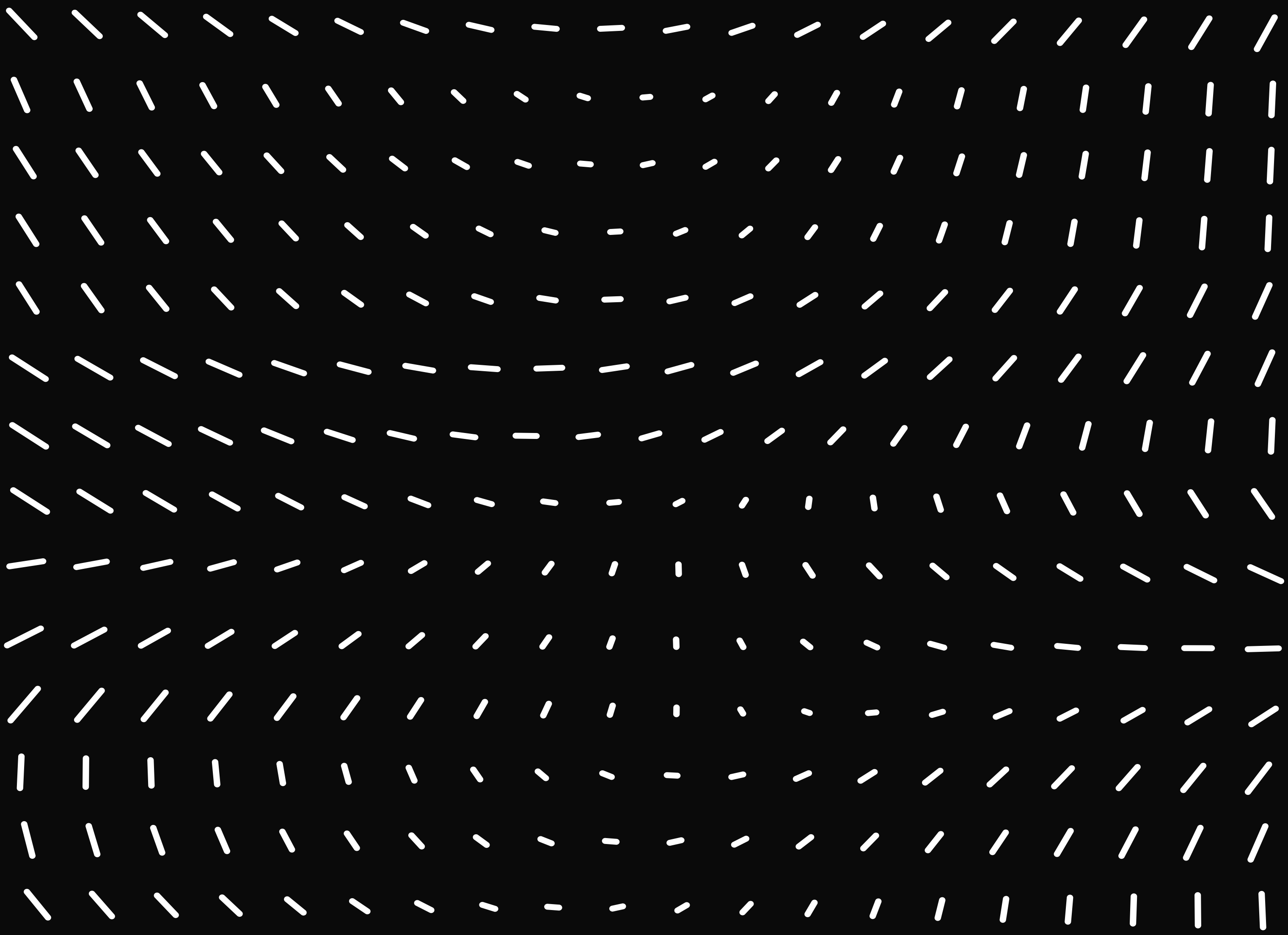
↓

The pair of points farthest from each other must be vertices of CH(Q)

33.4-5: — — —

4.1-5:

```
Max num = -∞ ;
Max pair = null;
While i <
```



Greedy Algorithm

↳ Greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimum choice in the hope that this choice will lead to a globally optimal solution.

问题1: Road Trip

*假设现在的目标是从 A 到 B，路上有 gas station，then 我们要如何规划路线才可以能到达目的地 and 去最少的加油站呢？

1. Sort the stations according to the distance from A

   ↳ $\{ \underbrace{S_0}_{\overset{\Downarrow}{A}}, S_1, S_2, \dots S_n, \underbrace{S_{n+1}}_{\overset{\Downarrow}{B}} \}$

2.

```
RoadTrip ( Si ) :      # Leaving at Si with a full tank
    t = i+1
    while  t < n  and  St+1 is reachable :
        t++                      ( 会在当 t == n 或 st+1 不能被 reach 时出 loop )
    if  t == n :
        stop          # we have arrived
    else :
        fill up gas at St
        Road Trip ( St )
```

Complexity: $\underbrace{O(n\log n)}_{\text{Sort}} + \underbrace{O(n)}_{\text{"actual" algorithm}} \Rightarrow O(n\log n)$

证明为什么这个 algorithm generate 出来的东西是 optimal 的

Assume $A = \{a_1, a_2 \dots a_n\}$ is a solution generated by this algorithm

Proof by induction:

Base case: if $n=0$，意即在出发地与目的地之间不存在任何 gas station，if there is a feasible solution (直接一罐油冲到底)，这个 algorithm 可以产生最优解

Assumption: 对于一个出发地与一个目的地之间，它们中间存在 x 个站 for which $x \le k$，algorithm 都能找到最优解

Proof: $k+1$:

假设此时，algorithm 找出的解是 $A = \{a_1, a_2 \dots\}$

optimal 找出的解是 $O = \{o_1, o_2, \dots\}$

因为算法的设计，所以车在到达 a_1 的时候会把油用完，所以 o_1 处于的位置必定不会在 a_1 之后，所以

o_1 在 a_1 之前 $\Longleftrightarrow$ o_1 到 o_2 有足够的油 $\Rightarrow$ a_1 到 o_2 必定有足够的油

↳ 上面这些证明了，把 a_1 替换掉 o_1 → $\{a_1, o_2, o_3 \dots\}$ 也是一个 optimal solution

since $|\{a_1, o_2 \dots\}| = |o_1, o_2 \dots|$

∴ a_1 is part of an optimal solution

∵ Assumption: $\{a_2, a_3 \dots\}$ is also an optimal solution

要把 a_1 和 $\{a_2, a_3 \cdots\}$ 结合在一起

↳ 那就考虑另一个问题：从加油站 a_1 出发到B，此时 $\{a_2, a_3 \cdots\}$ 和 $\{O_2, O_3 \cdots\}$ 都是这个问题的optimal solution

↳ 那么 因为同时加入了 a_1 之后，$|A| = |O^*|$，所以 $|A|$是optimal solution

↳ Algorithm finds an optimal solution for k+1 个 gas station, Proved!

问题 2： Activity-selection problem

* 假设现在有 n个 activity $\{a_1, a_2 \cdots a_n\}$ 都希望在同一个地点举行，但它们无法同时举行，设计一种算法 找出一种能容纳下尽可能多的 activity

* s_i 和 f_i denote the start time and finish time

先把全部 activity 给 sort 好（根据它们的 finish time）

那么现在整个算法的雏形已经出来了①先选择一个最早结束的activity (the first place of the sorted array)

②再选择下一个早结束的（但需要确保不 overlap）

Pseudo-code algorithm:

```
Activity = {a_1, a_2 .... a_n}   # sorted array
ActivitySelection(Activity):
    Select a_1
    current_time = a_1_finish_time
    For i=2 to n:
        if a_i_start_time >= current_time:
            select a_i
            current_time = a_i_finish_time
```

Complexity: $\underbrace{O(n\log n)}_{\text{merge sort}} + \underbrace{O(n)}_{\text{the 'actual' algorithm}} \Rightarrow O(n\log n)$

Proof by induction:

Base Case: if $n=0$，算法不会选择任何活动，因为根本没有活动可以选择！

Assumption: for $n=k$ (有k个活动), Activity Selection finds an optimal solution for x st $x \leq k$

k+1 :

假设 k+1数量的 array 丢进 ActivitySelection $\Rightarrow \{a_1, a_2 \cdots\}$

optimal solution $\Rightarrow \{o_1, o_2, \cdots\}$

因为算法本身的设计：a_1 必定是早结束的 $\Rightarrow$ o_1 在 a_1 的后面

在这种情况下，如果把 a_1 换进 o_1 $\Rightarrow |\{a_1, o_2, o_3 \cdots\}| = |\{o_1, o_2, o_3 \cdots\}|$

↳ a_1 is part of an optimal solution

又∵ $\{a_2 \cdots a_k\}$ is definitely an optimal solution (Thanks to assumption)

↳ 现在的问题 变成 了怎么把 $\{a_1\}$ 和 $\{a_2 \cdots a_k\}$ merge 起来

↳ Reduced problem：初始时间是 a_1 的 finish time，那么

$\{a_2, a_3 \cdots a_k\}$ 和 $\{o_2, o_3, \cdots o_k\}$ 都是 optimal solution

↳ 同时加入了 a_1 之后

$|A| = |O^*|$

∴ A也是 optimal solution

finds optimal solution for k+1 activities ⇒ Proved!

What if we sorted the array with start time?

$A: \{a_1, a_2, a_3 \cdots a_n\}$ $O = \{o_1, o_2, \cdots o_n\}$

a_1 必定早于 o_1 开始，但如果 o_1 finish 地比 a_1 早，那么刚刚那种方法会完全用不了

问题三： Coin Change

假设目前有钱币 ： {penny, nickel, dime, quarter, loonie, toonie}

面值(cents) ： {1, 5, 10, 25, 100, 200}

怎么能用最少(数量)的钱币 凑出值为 r 的总量呢？

↳ 那么一个 greedy 版本的算法已经油然而生了，对于总量 r，每次都找比 r 小，但最大的 coin，直至 r = 0

Pseudo - code :

```
首先 :  Coins = { v1, v2, v3 ... vk }  #硬币根据面值 大小 降序排列
        MinCoins (m) :          # m代表了目前需找的 target value
            r = m               # r 代表了 remaining amount, 一开始什么都没选，所以 r = m
            while r > 0 :
                find the maximal coin of value vi < r     # 因为 k是 constant, 所以这里是 O(1)
                Select a coin of value vi                  # 还可以理解成 放进来的已经 sort好了,
                r = r - vi                                 # m 的 成长与其无关
```

Proof:

Base case : 有 NMD 五个

① For $1 \leq m < 5$: optimal solution : m个 pennies (1)

algorithm solution : m个 pennies (1) ✓

For $m = 5$: algorithm choose only 1 nickel (5) ✓

② For $5 < m < 10$: Only solution available is m pennies(1) or 1 nickel (5) + m-5个 pennies(1)

algorithm solution : 1 nickel (5) + m-5个 pennies (1) ✓

For $m = 10$: algorithm choose only 1 nickel (5) ✓

③ For $10 < m < 25$: 在这里面的 optimal solution 不可能有超过 4个 pennies (1) (因为 5个 pennies 可以被 1个 nickel 换掉)，同理，不能存在超过一个 nickel(5) (2个 nickel(5) 可以被一个 dime (10) 换掉)

但这些 most total = (4×1 + 1×5) = 9，所以超过 9的值 必定得存在 一个 dime (10)

当 $10 < m < 25$ 选择了一个 dime 之后，$m - 10 = m'$

↙ ↓

$0 < m' < 10$ → proved

$10 < m' < 15$ → 再选一个 coin → $m' - 10 = m''$ → $0 < m'' < 5$ proved

慢慢这样证下去，我们一直证到 200

当 $n \leq 200$, algorithm finds optimal solution

Assumption : the algorithm finds optimal solution for n such that $200 < n \leq k$

现在证明 K+1

MinCoin (K+1) = $\{a_1, a_2, a_3 \cdots\}$ a_1是 tonnie (200)

optimal = $\{o_1, o_2, o_3 \cdots\}$

Suppose 在 optimal 中，不存在 tonnie (200) ，即不存在 a_1

如果没有 tonnie (200) ，用刚刚证明 $10 \le m < 25$ 的思想，optimal solution can contain at most

loonie (100) ×1

quater (25) ×3

pennies (1) × 4

dimes (10) ×2 or dime (10) ×1

nickel (5) ×1

这边多有一个 nickel (5) 可被 quater 换掉

这边多给一个 dime (10) 可被 quater 换换

两边 sum up 199 194 → 都不满足，所以在 optimal solution 中必定存在 a_1 (tonnie) (200)

然后因为 $m-a_1$之后，它的值落在了 $0 \le m < k$ 之间，which is optimal by assumption

∴ a_1 is part of an optimal solution

$\{m-a_1\}$ finds optimal solution O^*

同时加上 a_1 → $|A| = |O^*|$ → proved !

Note：并不是所有 possible 面值都可以用贪算法得出正确答案

Example: $\{1, 4, 9\}$, $m=12$

algorithm solution : 9, 1, 1, 1,

optimal solution : 4, 4, 4

问题4: Knapsack Problems

↳ Given a Container of capacity K and a set of items $\{a_1, \dots, a_n\}$, each of which has mass m_i and value V_i, we call a subset $S \subseteq A$ feasible if $\sum_{a_i \in S} m_i \le K$, Our goal is to find a feasible subset s^* that maximizes $\sum_{a_i \in S^*} V_i$.

⇑ 这是个 无法用 Polynomial-time algorithm 解决的 问题

所以, 问题被 modified 成了 ⇓

↳ Given a Container of capacity K and a set of items $\{a_1, \dots, a_n\}$, each of which has mass m_i and value V_i, we call a subset $S \subseteq A$ feasible if $\sum_{a_i \in S} m_i \le K$, Our goal is to find a feasible subset s^* that maximizes $\sum V_i$. Allowing fractions of objects to be used, where the value of a fraction of an object is the same fraction of the value of the object

More formally: Given k and a set of n pairs of the form (v_i, m_i) find a set of values $\{p_1, p_2, \dots p_n\}$ such that $0 \le p_i \le 1 \quad \forall i$ and $\sum_{i=1}^{n}(p_i \cdot m_i) \le k$ and $\sum_{i=1}^{n}(p_i \cdot v_i)$ is maximized

Greedy FKS:

```
Sort the objects in decreasing $\frac{V_i}{m_i}$ order

while $K > 0$ and there are still objects to consider:
    Take as much of the next item as possible
    Reduce K by the mass amount just added to the Knapsack
```

比如 现只剩下 2 的空位, Queue 里面轮到了 x_3

把2全装, 拿走了 $\frac{2}{20} = \frac{1}{10}$ 个 x_3 ⇐ $\begin{cases} \text{value} : 30 \\ \text{mass} : 20 \end{cases}$

价值 $+= \frac{1}{10} \times 30 = 3$

证明: For Base case: 0个 object: 什么都选不了, 是 optimal

Assumption: FKS finds the optimal solution for x 个 object such that $x \le K$

$K+1$: object: $\{x_1, x_2, x_3 \cdots\cdots x_{K+1}\}$

FKS find: $\{p_1, p_2, p_3, \cdots\cdots p_{K+1}\}$ # p_1 代表了 portion of x_1

Optimal : $\{q_1, q_2, q_3, \cdots\cdots q_{K+1}\}$

因为算法的设计的缘故, 所以 p_1 肯定是尽可能地大, 所以 suppose $p_1 > q_1$

又∵ total_mass (A) = total_mass (o)

∴ 必定存在 i where $i > 1$ where $p_i < q_i$ for x_i

Let $O^* = \{\underline{p_1}, q_2, \cdots q_i', \cdots, q_{K+1}\}$ ⇒ $q_i' = q_i - (p_1 - q_1)$ # 把 x_1 的占比挪回来 $\quad (p_1 - q_1)\, x_1\text{ mass} = (q_i - q_i')\, x_i\text{mass}$

⇓

$(p_1 - q_1) = q_i - q_i'$

之前 q_i 比 p_i 多的

$$\text{total\_value}(O^*) - \text{total\_value}(o) = (p_1 - q_1)\, x_1.\text{value} - (\underline{q_i - q_i'})\, x_i.\text{value}$$

$$= (p_1 - q_1)\, x_1.\text{value} - (p_1 - q_1)\frac{x_1\text{ mass}}{x_i\text{ mass}}\, x_i\text{ value}$$

$$= (p_1 - q_1) \cdot x_1\text{ mass} \cdot \left(\frac{x_1.\text{value}}{x_1\text{ mass}} - \frac{x_i\text{ value}}{x_i\text{mass}}\right) > 0$$

分部拆解:

$$O = \{q_1, q_2, q_3, \cdots q_i, \cdots q_{k+1}\}$$

$$O^* = \{p_1, q_2, q_3, \cdots q_i', \cdots q_{k+1}\} \Rightarrow \begin{cases} p_1 = q_1 + (q_i - q_i') \\ (p_1 - q_1)\, x_1\, mass = (q_i - q_i')\, x_i\, mass \end{cases}$$

$$(q_i - q_i') = (p_1 - q_1) \frac{x_1 mass}{x_i mass}$$

$$total\_mass(O) = total\_mass(O^*)$$

$$total\_value(O^*) - total\_value(O) = (p_1 - q_1)\, x_1.value + (q_i' - q_i)\, x_i.value$$

$$= (p_1 - q_1)\, x_1 value + q_i' x_i value - q_i x_i value$$

$$= (p_1 - q_1)\, x_1 value - q_i x_i value + q_i' x_i value$$

$$+ q_i' x_i value - q_i x_i value$$

$$= (p_1 - q_1)\, x_1 value - (q_i - q_i')\, x_i value$$

$$= (p_1 - q_1)\, x_1 value - (p_1 - q_1) \frac{x_1 mass}{x_i mass}\, x_i value$$

$$= (p_1 - q_1) \left(x_1 value - \frac{x_1 mass}{x_i mass}\, x_i value\right)$$

$$- q_i x_i value +$$

$$= (p_1 - q_1)\, x_1 mass \left(\frac{x_1 value}{x_1 mass} - \frac{x_i value}{x_i mass}\right)$$

$$\because \frac{x_1 value}{x_1 mass} > \frac{x_i value}{x_i mass} \Rightarrow total\_value(O^*) > total\_value(O)$$

↳

并且可以知道的是，在 O^* 中 x_1 的占比和 A 中 x_1 的占比是一样的 ($p_1 = p_1$)

↳ S_A 和 O^* has one fewer difference than S_A and O did.

↳ 不停地重复 until we reach a state where $S_A = O$……

So S_A is optimal

问题五： Huffman Encoding

↳ 比如现在要Encode 一串 string，那么每个字母都有自己的 code

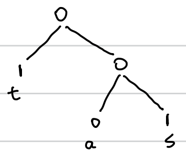
例: Alphabet： A B C

code： 001 010 100

↳ 要怎么做才可以提高效率呢？ → 用更短的 bit，去表示一样的意思

↳ * 给每个字母不一样长度的 bit representation → 用的更多的可以用少一点 bit，用的少的可以用多一点的 bit

↳ * 同时需要知道的是，用 tree 可以代表 string representation



⇒ t a s

01 000 001

那么 greedy algorithm的大意就出来了：

① Sort the alphabet base on frequency , low frequency 在前

② Repeat until element in alphabet >1 :

取出 frequency最低的两个 element from alphabet

make them as leaf nodes and apply value 1 or 0

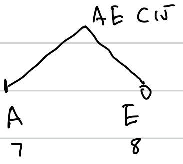
Replace those two elements with one combined elements in alphabet

O(nlogn)

Example：

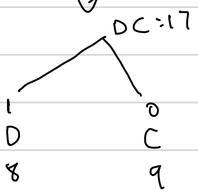
S= {A, E, D, C, F, B} (sorted)

7 8 8 9 11 16



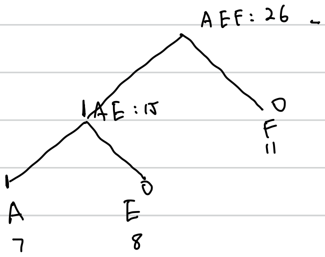
S'={D, C, F, AE, B}

8 9 11 15 16



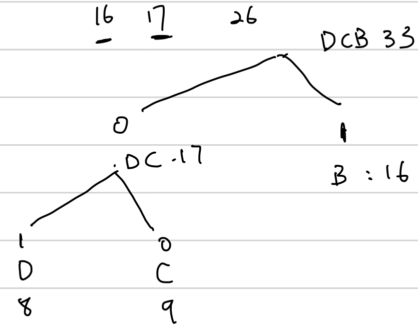
S''={F, AE, B, DC}

11 15 16 17



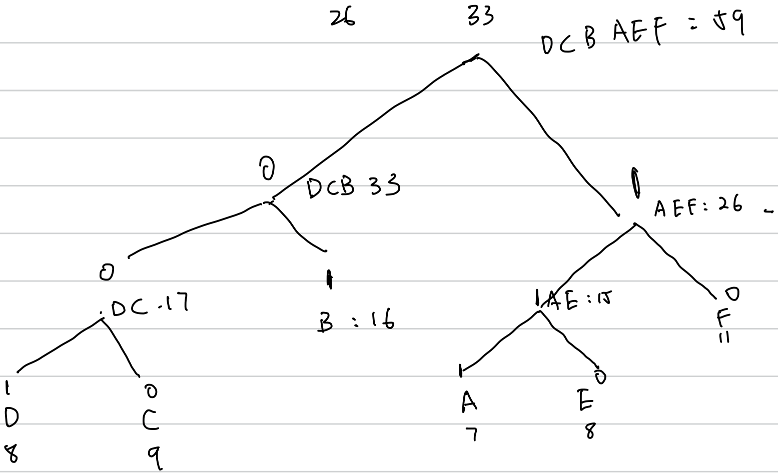
S''' ={B, DC, AEF}

16 17 26



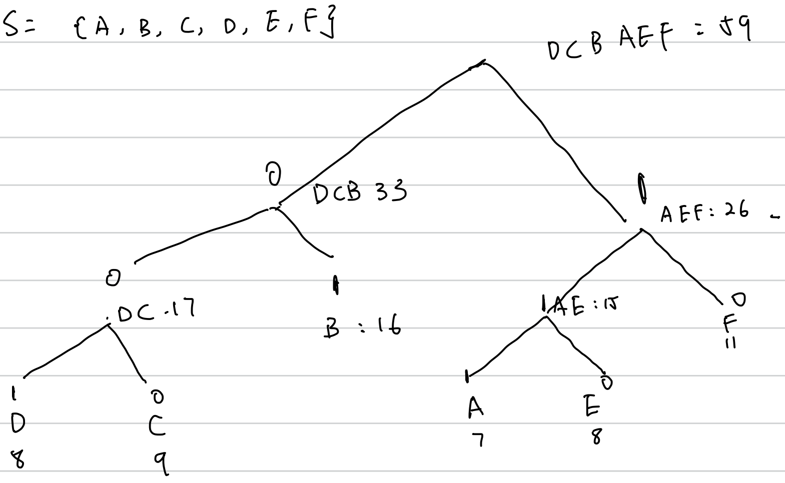
S'''' ={AEF , DCB}

26 33



很重要的问题：如何判断好坏

$S = \{A, B, C, D, E, F\}$



	Frequency	code	length
A	7	111	21
B	16	01	32
C	9	001	27
D	8	001	24
E	8	110	24
F	11	10	22

总 bits = $\sum_{i=0}^{i<len(S)}$ frequency * |code|

∴ 总 bits = 150 bits

↓
普通 ASCii code

	Frequency	code	length
A	7	000	21
B	16	001	48
C	9	010	27
D	8	011	24
E	8	100	24
F	11	101	33

总 bits = $\sum_{i=0}^{i<len(S)}$ frequency * |code|

∴ 总 bits = 177 bits

*并且 depth of a node equals to the number of bits in its bit representation

Proof:

Base case: when $n=0$ (没有字母), do nothing

Assumption: Assume that for $n = x$ (有 x 个字母), algorithm finds the optimal solution ($x <= K$)

Proof (K+1): $S = \{x_1, x_2, x_3, \cdots, x_K, x_{K+1}\}$

Algorith finds $\{a_1, a_2, a_3, \cdots a_K, a_{K+1}\}$

Optimal solution $\{O_1, O_2, O_3, \cdots O_K, O_{K+1}\}$

因为算法的特性，x_1 是 frequency 最低的，所以 x_1 必定在 tree 的最底层

又∵ Depth (x) = length for representation of x

∴ Assume $a_1 \neq O_1$

那么 $a_1 > O_1$

∵ This is a prefix code，在 O 中，必定存在 O_i (where $i > 1$) 在 tree 的最底层

构建 O^*，such that O_1 和 O_i 交换位置 => $\{O_i, O_2, O_3, \cdots O_1, \cdots O_K, O_{K+1}\}$

total (O^*) − total (O) = (x_1.frequency * $|O_i|$ − x_1.frequency * $|O_1|$) + (x_i.frequency * $|O_i|$ − x_i.frequency * $|O_1|$)

= x_1.frequency * $|O_i|$ − x_1.frequency * $|O_1|$ + x_i.frequency * $|O_i|$ − x_i.frequency * $|O_1|$

= x_1.frequency * ($|O_i| - |O_1|$) + x_i.frequency ($|O_i| - |O_1|$) > 0

$|O_i| > |O_1|$ 因为假设 x_i 在最底层

∴ x_1 应该在算法的最底层，

∴ a_1 is part of an optimal solution

Huffman Code

Proof:

Base case: when $n=1$ (一个字母), 1 bit 代表

Assumption: Assume that for $n=x$ (有x个字母), algorithm finds the optimal solution ($x<=K$)

Proof ($K+1$): $S=\{x_1, x_2, x_3 \cdots, x_K, x_{K+1}\}$

Algorith finds $\{a_1, a_2, a_3, \cdots a_K, a_{K+1}\}$

Optimal solution $\{O_1, O_2, O_3, \cdots O_K, O_{K+1}\}$

因为算法的特性, x_1是 frequency 最低的, 所以 x_1 必定在 tree 的最底层

又∵ Depth (x) = length for representation of x

∴ Assume $a_1 \neq O_1$

那么 $a_1 > O_1$

∵ This is a prefix code, 在 O 中, 必定存在 O_i (where $i>1$) 在 tree 的最底层

构建 O^*, such that O_1 和 O_i 交换位置 ~~=> {~~$O_i, O_2, O_3, \cdots O_1, \cdots O_K, O_{K+1}\}$

$\text{total}(O^*) - \text{total}(O) = (x_1.\text{frequency} * |O_i| - x_1.\text{frequency} * |O_1|) + (x_i.\text{frequency} * |O_1| - x_i.\text{frequency} * |O_i|)$

$= x_1.\text{frequency} * |O_i| - x_1.\text{frequency} * |O_1| + x_i.\text{frequency} * |O_i| - x_i.\text{frequency} * |O_1|$

$= x_1.\text{frequency} * (|O_i| - |O_1|) + x_i.\text{frequency}(|O_i| - |O_1|) > 0$

$|O_i| > |O_1|$ 因为假设 x_i 在最底层

∴ x_1 应该在算法的最底层,

∴ a_1 is part of an optimal solution

然后可以用同样的方法来证明 a_2 is part of the optimal solution

∴ x_1, x_2 两个 frequency 最低的在同一最低级 → Reduced problem of $K-2$

然后可以说, By Hypothesis, for K 个字母, algorithm finds optimal solution, by adding x_1, x_2 in it, both O and A will put them in the same level, which means that $|A| = |O|$, A is the optimal solution

$x_1.\text{frequency} * |O_i| - x_1.\text{frequency} * |O_1|) + x_i.\text{frequency} * |O_1| - x_i.\text{frequency} * |O_i|$

$+$

$= x_1.\text{frequency} * (|a_1| - |O_1|) + x_i.\text{frequency} * (|O_1| - |a_1|)$

$=$

Huffman Code

Proof:

Base case: when $n=1$ (一个字母), 1 bit 代表

Assumption: Assume that for $n=x$ (有x个字母), algorithm finds the optimal solution ($x \le k$)

Proof (k+1): $S = \{x_1, x_2, x_3, \cdots, x_k, x_{k+1}\}$

Algorith finds $\{a_1, a_2, a_3, \cdots a_k, a_{k+1}\}$

Optimal solution $\{O_1, O_2, O_3, \cdots O_k, O_{k+1}\}$

因为算法的特性, x_1, x_2 是 frequency lowest, therefore those two will be put in the level with the largest depth it the tree

∵ depth (node) is the length of bit representation of the node

Suppose $a_1 \neq a_2 \neq O_1 \neq O_2$

∴ depth (a_1), depth (a_2) are already two alphabet with largest depth

∴ O_1, O_2 has length smaller than a_1, a_2

∴ It's a prefix code

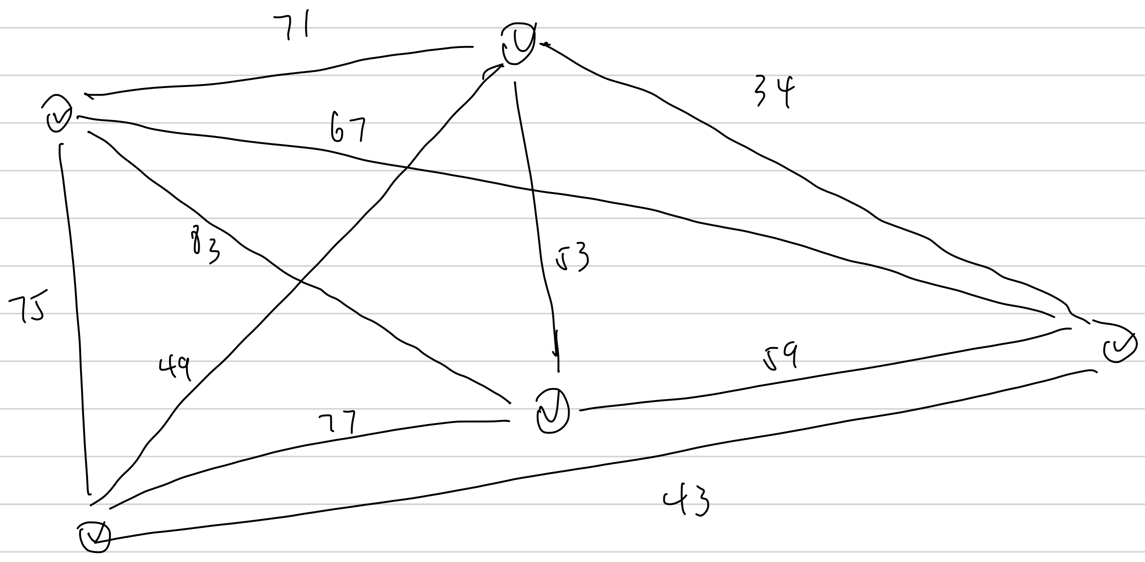
There exist O_i, O_j where $i > 1$, $j > 1$ will be placed in the deepese of the tree in optimal solution

Construct O^* such that $\{O_i, O_j, O_3, O_4 \cdots O_1, O_2 \cdots\}$ # Swap O_i, O_j in the tree

$$\text{total\_value}(O^*) - \text{total\_value}(O)$$
$$= |O_i| * (x_1.\text{frequency}) + |O_j| * (x_1.\text{frequency}) - (|O_1| * (x_i.\text{frequency}) + |O_2| * (x_i.\text{frequency}))$$
$$= |O_i| * (x_1.\text{frequency}) + |O_j| * (x_1.\text{frequency}) - |O_1| * (x_i.\text{frequency}) - |O_2| * (x_i.\text{frequency})$$

$$= (x_1.\text{frequency})(|O_i| + |O_j|) - x_i.\text{frequency}(|O_1| + |O_2|)$$

34 +43 + 53 + 67



然后可以用同样的方法来证明 a_2 is part of the optimal solution

∴ x_1, x_2 两个 frequency 最低的在同一最低级 → Reduced problem of K-2

然后可以说, By Hypothesis, For k 个字母, algorithm finds optimal solution, by adding x_1, x_2 in it, both O and A will put them in the same level, which means that $|A| = |O|$, A is the optimal solution

Problem: 胡老师

① Can you convert the recursive greedy algorithm Road Trip to an iterative algorithm?

② Text book P422, Exercise 16.1-2

③ Text book P422, Exercise 16.1-3

④ Design and prove a greedy algorithm for coin change when use American Coins

⑤ Make example and show that greedy strategy does not hold for the 0/1 knapsack Problem?

⑥ Write the Pseudocode and implement the Huffman coding greedy algorithm

罗老师

$$2^{32} \times 2^{2} \times 2^{10}$$

$$= 2^{\boxed{20}} \longrightarrow$$

16.1-2

Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

16.1-3

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

16.1-4

Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This problem is also known as the ***interval-graph coloring problem***. We can create an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices have the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

16.1-5

Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

16.1-2:

This is a greedy algorithm because algorithm always looks for best solution (The last activity to start) currently.

Base case: Only one activity, choose 1

Induction Hypothesis: Algorithm finds the optimal Solution for $n \leq k$ activities

k+1: suppose algorithm finds $\{a_1, a_2, \cdots a_k, a_{k+1}\}$ optimal solution is $\{o_1, o_2, \cdots o_k, o_{k+1}\}$

Suppose $a_1 \neq o_1$, based on the design of algorithm, a_1 is the last activity to start, there for start time for o_1 is earlier than a_1

o_1 and o_2 is compactable because o_1 start after o_2 finish, so a_1 and o_2 is compactable. Therefore, if we construct a o^* such that $o^* = \{a_1, o_2, o_3 \cdots o_{k+1}\}$

$|o^*| = |o|$, → o^* is optimal, a_1 is part of an optimal solution ...

16.1-5

16.1-3

based on duration of activity:

$$\left\{\begin{array}{ll} 1 - 4 & (3) \\ 4 - 5 & (1) \\ 4 - 7 & (3) \end{array}\right.$$

algorithm: (4-5)

optimal: (1-4) (4-7)

based on overlaps of fewest of other remaining

$$\left\{\begin{array}{l} 10 - 17 \\ 1 - 5 \\ 6 - 8 \\ 4 - 7 \end{array}\right.$$

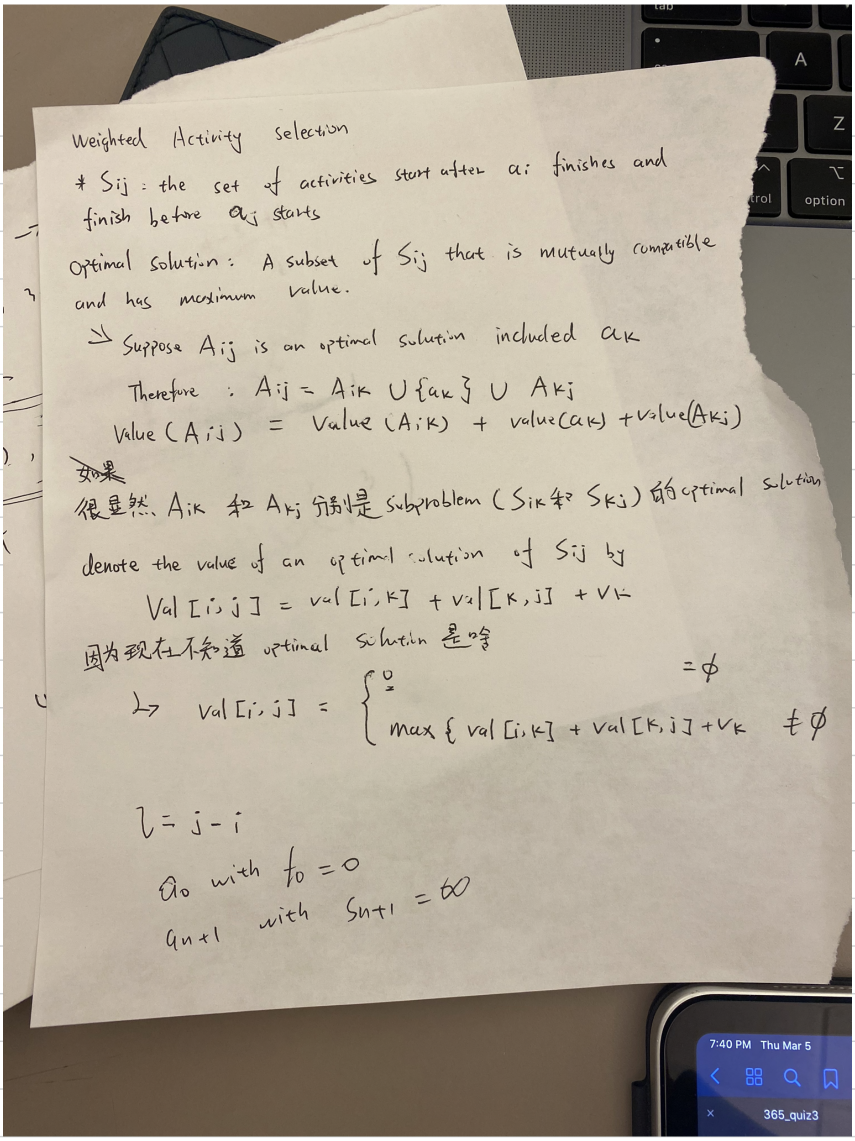
algorithm: 10 - 17

optimal: (1-5, (6-8)

16.1-4:

```
1. Sort activities base on Start time
2. i=0
3. k=0
4. while i < len (activity):
    if there exist Room[k]. finish_time <= activity[i]. start_time:
            put activity [i] into Room[k]
            Room[k]. finish_time = activity[i]. finish_time
    else:
            k += 1
            Room[k] = new_room
            put activity [i] into Room[k]
            Room[k]. finish_time = activity[i]. finish_time
```

X → 要检查所有的房间



Weighted Activity selection

* S_{ij} : the set of activities start after a_i finishes and finish before a_j starts

Optimal solution : A subset of S_{ij} that is mutually compatible and has maximum value.

⇒ Suppose A_{ij} is an optimal solution included a_K

Therefore : $A_{ij} = A_{iK} \cup \{a_K\} \cup A_{Kj}$

$\text{Value}(A_{ij}) = \text{Value}(A_{iK}) + \text{value}(a_K) + \text{value}(A_{Kj})$

~~如果~~

很显然, A_{iK} 和 A_{Kj} 分别是 subproblem (S_{iK} 和 S_{Kj}) 的 optimal solution

denote the value of an optimal solution of S_{ij} by

$\text{Val}[i,j] = \text{val}[i,K] + \text{val}[K,j] + v_K$

因为现在不知道 optimal solution 是啥

$$\hookrightarrow \text{val}[i,j] = \begin{cases} 0 & = \phi \\ \max\{\text{val}[i,K] + \text{val}[K,j] + v_K & \neq \phi \end{cases}$$

$l = j - i$

a_0 with $f_0 = 0$

a_{n+1} with $S_{n+1} = \infty$

```
MAX-VALUE-ACTIVITY-SELECTOR$(s, f, v, n)$
  let $val[0..n+1, 0..n+1]$ and $act[0..n+1, 0..n+1]$ be new tables
  for $i = 0$ to $n$
      $val[i, i] = 0$
      $val[i, i+1] = 0$
  $val[n+1, n+1] = 0$
  for $l = 2$ to $n+1$
      for $i = 0$ to $n - l + 1$
          $j = i + l$
          $val[i, j] = 0$
          $k = j - 1$
          while $f[i] < f[k]$
              if $f[i] \le s[k]$ and $f[k] \le s[j]$ and
                          $val[i, k] + val[k, j] + v_k > val[i, j]$
                  $val[i, j] = val[i, k] + val[k, j] + v_k$
                  $act[i, j] = k$
              $k = k - 1$
  print "A maximum-value set of mutually compatible activities has value "
          $val[0, n+1]$
  print "The set contains "
  PRINT-ACTIVITIES$(val, act, 0, n+1)$
```

CISC/CMPE-365*
Test #2
October 18, 2013

Student Number (Required) ______________________

Name (Optional)________________________________

This is a closed book test. You may not refer to any resources.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, but will not be reconsidered after the test papers have been returned.

The test will be marked out of 50.

Question 1	/25
Question 2	/20
Question 3	/5
TOTAL	/50

The point is, ladies and gentleman, greed is good. Greed works, greed is right. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. Greed in all its forms, greed for life, money, love, knowledge has marked the upward surge in mankind.

— Michael Douglas as Gordon Gekko,
Wall Street
(1987)

General Marking Instructions:

My general philosophy is that students should only fail a test if it is clear that they made no effort at all to prepare for it. This sets the bar quite low for passing. However, I believe that I set fairly difficult exams.

Please don't give 0 points for any question unless the student leaves the page blank or writes something completely unrelated to the question. Even if what they write is only marginally related to the proper answer, please give them 1 or 2 marks.

Students who show an understanding of the question should get at least 50% on the question, even if they are unable to answer it.

If a student writes enough to show that they know what to do to answer the question, even though they can't complete it, should get about 75% on the question.

For example, suppose the question is "Show that problem X is NP-Complete."

A student who answers, "I need to show that X is in NP, and I need to show that all problems in NP reduce to X" or something similar should get about 50%

A student who answers, "I need to show that X is in NP, and I need to show that instances of some known NP-Complete problem Y can be transformed in polynomial time into instances of X, in an answer-preserving way." or something similar should get about 75%

A student who identifies an appropriate NP-Complete problem Y and has some idea about the transformation should get about 80% ...

... and so on

Beyond that, take off a mark or two for significant errors or omissions. If a

student gives an answer that is completely correct except for a trivial error, you can give full marks or take 0.5 off – it's up to you, as long as you are consistent.

Question 1 (25 marks)

In Elbonia the only coins have value 1, 5 and 10 kronks. The inhabitants use the following (obvious) greedy algorithm to choose coins to add up to a given target value:

Let k be the desired total value.

```
while k >= 10:
     take a 10-kronk coin
     k = k - 10
while k >= 5:
     take a 5-kronk coin
     k = k - 5
take k 1-kronk coins
```

Prove that this algorithm always uses the smallest number of coins to add up to any target value of k >= 1.

Hint: Start by showing that it works for all k < 10, then use induction.

Proof:

Base case: when $1 \leq n < 5$: optimal : $1*n$, which is excatly what the algorithm does

when $5 \leq n < 10$:

there are two solutions : $1*n$ & $5*1 + (n-5)*1$ (Algorithm does)

Therefore clearly Algorithm finds the optimal solution

$\therefore$ when $n < 10$, algorithm finds an optimal solution

Hypothesis : For $10 < n \leq K$, algorithm finds the optimal solution

Proof K+1 :

↳ For value of K+1, $K+1 > 10$, algorithm finds a 10 Kronks

Assume that the optimal solution doesn't exist a 10 Kronk

However, there are at most 4 1 Kronk, 1 5 Kronk, the total value is 9 which doesnt satisfy the requirement. Therefore 10 Kronks definitely exist in the optimal solution

↳ the value reduce to $K+1 - 10 = K-9$

$\because K-9 < K$, By Hypothesis, Algorithm finds the optimal solution for $n \leq K$

$\therefore$ Algorithm finds optimal for $(K-9)$

when we add $(K-9)$ by 10, total number of coin will only be increased by one

$\therefore$ A is the optimal solution

(Blank page if needed for answering Question 1)

Solution: Consider n <= 4. The only solution is to take n 1-kronk coins, which is what the algorithm does. Consider 5 <= n < 10. The only possible solutions are A={n 1-kronk coins} and B={1 5-kronk + n-5 1-kronk coins}. Clearly B is better (n-4 < n), and this is what the algorithm does.

IA: Suppose the algorithm finds an optimal solution whenever the target value is < n, for some n >= 10.

Let the target value be n. The algorithm starts by taking 1 10-kronk coin. We will prove that there is an optimal solution that contains at least 1 10-kronk coin. Let O be an optimal solution for n, such that O does not contain any 10-kronk coins. If O contains 5 or more 1-kronk coins O cannot be optimal, since 5 of these can be replaced by 1 5-k. coin. If O contains 2 or more 5-k coins) cannot be optimal since 2 of these can be replaced by 1 10-k coin. Thus O contains at most 1 5-k and 4 1-k coins. But then n < 10, which is a contradiction. Thus O contains at least 1 10-k coin, so the algorithm's first choice is correct.

After the first choice is made, the target value is reduced to < n. By the IA, the algorithm finds an optimal solution to the reduced problem. Applying the standard argument, let O be an optimal solution that starts with a 10-k coin. The rest of O solves exactly the same reduced problem as the reduced problem the algorithm solves optimally. This part of O must have exactly the same size as this part of the algorithm's solution, so the size of the algorithm's complete solution equals the size of O. Therefore the algorithm's solution is optimal when the target is n.

Therefore the algorithm's solutions is optimal for all n.

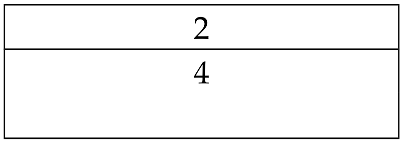
Other proofs are possible and acceptable.

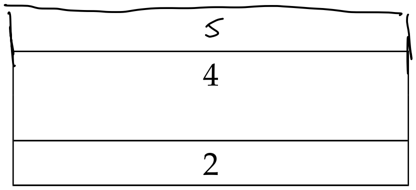
Students often have great difficulty giving correct proofs. Please be kind :)

Question 2 (20 Marks)

Suppose we have **n** concrete blocks, each with a certain thickness (i.e. block **i** has thickness **t**$_i$). We need to stack the blocks into a single stack. Clearly the total height of the stack will be the same no matter what order we stack the blocks. However, the **sum** of the elevation above the ground of the tops of the blocks will depend on the order in which we stack the blocks.

Example: if there are only two blocks, with thickness 2 and 4, then stacking the 2 on top of the 4 will give a sum of elevations of 4 + 6 = 10, but stacking the 4 on top of the 2 will give a sum of elevations of 2 + 6 = 8.





2 + 6 + 11

(a) **(5 marks)** Create a Greedy Algorithm to find the order in which the blocks should be stacked to **minimize** the average height of the tops of the blocks.

[redacted]

[redacted]

[redacted]

```
Min Stack ( T )
    Sort all the blocks by ti with increasing order
    stair = null
    i = 0
    while i < len(T) :
        Stack T[i] on stair
        stair = T[i]
        i ++
```

(b) **(15 marks)** Prove that your algorithm is correct.

Solution:

Observe that the set of possible solution values is non-empty and finite, and must have a minimum element, so an optimal solution does exist.

An inductive proof similar to Question 1 is perfectly acceptable.

OR a proof along these lines:

Let the blocks be b1 <= b2 <= b3 <= ... <= bn

The algorithm's solution is to stack them in this order – call this order A

Choose O from the set of optimal solutions such that O has the greatest agreement with A, starting from the beginning.

Suppose that O differs from A. Let i be the position of the first difference. That is

A = b1, b2, ...b(i-1), bi, ... bn

O = b1, b2, ... b(i-1), x, ..., bi, ... where x != bi (note that bi must be after x in O, and x >= bi)

Let O' = b1, b2, ... b(i-1), bi, ..., x, ... (ie exchange x and bi in O)

Note that the elevation of each of b1, ... b(i-1) does not change
Note that the elevation of each block on top of x in O' does not change
Note that the elevation of x is now the same as the elevation of bi in O
Note that the elevation of bi is now <= the elevation of x in O
Note that the elevation of each block between bi and x in O' is <= its elevation in O

Thus the sum of elevations in O' is <= the sum of elevations in O. Thus O' is optimal, and it has greater agreement with A than O did. Contradiction.

Therefore O does not differ from A – ie A is optimal.

Proof:

Base Case: For $n=0$ (0 blocks), algorinthm stack nothing

Hypothesis: for $n <= K$, MinStack (T) finds optimal solution

K+1: For T of len (K+1) $= \{x_1, x_2, x_3, x_K, x_{K+1}\}$

Min Stack (T) $= \{a_1, a_2, a_3 \ldots\ldots a_K, a_{K+1}\}$

Optimal solution : $\{O_1, O_2, O_3 \ldots\ldots, O_K, O_{K+1}\}$

Algorithm always finds the least thickness block as the first one

∴ Assume $a_1 \neq O_1$, O_1 is thicker than a_1

∴ The position of a_1 exist at some other point O_i where $i>1$

Construct O^* such that $O^* = \{a_1, O_2, O_3 \ldots\ldots O_1, \ldots O_K, O_{K+1}\}$ # Snap the position for O_1 and O_i

$$\text{Evaluation}(O) - \text{Evaluation}(O^*) = O_1 + (O_1+O_2) + (O_1+O_2+O_3) + \ldots\ldots + (O_1+O_2+O_3+\cdots O_{K+1})$$

$$-(a_1 + (a_1+O_2) + (a_1+O_2+O_3) + \ldots\ldots + (a_1+O_2+O_3+\cdots O_{K+1})\ )$$

$$= K(O_1 - a_1) > 0 \quad ∵ \ O_1 \text{ is thicker than } a_1$$

∴ We can construct a better solution O^* such that a_1 is at first position

∴ a_1 is part of an optimal solution

By Induction Hypothesis: Algorithm finds the optimal solution, consider a reduced problem that we only have $\{a_2, a_3 \ldots\ldots a_{K+1}\}$ block, in this case, $A = O$

∴ Both optimal solution and

CISC/CMPE-365*
Test #2
October 17, 2014

Solutions and Marking Guide

This is a closed book test. You may not refer to any resources.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, **but will not be re-marked under any circumstances.**

The test will be marked out of 50.

Question 1	/20
Question 2	/10
Question 3	/20
TOTAL	/50

"Do the unexpected."
Happy Birthday to Rick Mercer

General marking philosophy: a student who gives enough of an answer to show they understood what they were supposed to do, even if they couldn't do it (or made lots of errors while doing it) should get at least 50% on that question.

Full marks should be given if a solution is sound and not missing anything important.

Feel free to give marks like 9.5/10 to a solution that is correct but contains a minor error.

Students may come up with solutions that are completely different from mine but still completely correct. Correct solutions should get full marks even if they don't match mine.

Students should always get a few marks for trying a question. The only way to get a 0 is to leave the page blank or write something completely irrelevant.

QUESTION 1 (20 Marks)

Suppose **Merge(List1, List2)** is a built-in function that takes two sorted lists and returns a new sorted list that combines the two original lists.

Merge(List1, List2) executes n copy operations, where n is the sum of the lengths of the two lists being merged.

You have been given k sorted lists L1, L2, … Lk. L1 has length n1, L2 has length n2, etc.

Your task is to use repeated calls to Merge() to create a single list that combines all the original lists.

For example, suppose there are three lists L1, L2, and L3, with n1 = 7, n2 = 4, n3 = 5. You could merge L1 with L2 (requiring 11 operations), then merge that combined list with L3, (requiring 16 operations), for a total of 27 operations.

Alternatively you could merge L2 with L3 (9 operations), then merge that combined list with L1 (16 operations), for a total of only 25 operations.

a) [10 marks] Create a Greedy Algorithm to merge the k lists using the **fewest possible** copy operations. Express your algorithm in clear pseudo-code.

A simple version of the solution looks like this:

```
while k > 1:
    let l_1 and l_2 be the two smallest lists
    l_1 = merge(l_1, l_2)
    remove l_2 from the set of lists
    k -= 1
```

There are many other ways to describe the process, but the key idea is to always merge the two smallest lists.

QUESTION 1 (20 Marks)

必须要先证明有 Solution

↓

有 optimal solution

Suppose **Merge(List1, List2)** is a built-in function that takes two sorted lists and returns a new sorted list that combines the two original lists.

Merge(List1, List2) executes n copy operations, where n is the sum of the lengths of the two lists being merged.

You have been given k sorted lists L1, L2, … Lk. L1 has length n1, L2 has length n2, etc.

Your task is to use repeated calls to Merge() to create a single list that combines all the original lists.

For example, suppose there are three lists L1, L2, and L3, with n1 = 7, n2 = 4, n3 = 5. You could merge L1 with L2 (requiring 11 operations), then merge that combined list with L3, (requiring 16 operations), for a total of 27 operations.

Alternatively you could merge L2 with L3 (9 operations), then merge that combined list with L1 (16 operations), for a total of only 25 operations.

a) [10 marks] Create a Greedy Algorithm to merge the k lists using the **fewest possible** copy operations. Express your algorithm in clear pseudo-code.

```
1、 Sort those lists by their length in increasing order
2、 i = 0
3、 Start = null
4、 while i < K :
        Start = ( start , T[i] )
        i++
```

Proof:

Base case : $n = 0$, then algorithm return null array

Induction Hypothesis : For $n \leq K$, algorithm finds the optimal solution

$K+1$:

Suppose algorithm produce : $\{a_1, a_2, a_3, \ldots a_k, a_{k+1}\}$

optimal solution : $\{O_1, O_2, O_3, \ldots O_k, O_{k+1}\}$

Suppose $A \neq O$, the first difference is at position i

↳ $A = \{a_1, a_2, a_3, \cdots a_{i-1}, a_i, x, \cdots a_k, a_{k+1}\}$ ← Swap the two position

$O = \{a_1, a_2, a_3, \ldots a_{i-1}, x, a_i, \ldots a_k, a_{k+1}\}$

Note that Every steps above x are the same in A

Every steps count below and included a_{i-1} are the same

∵ Base on algorithm, a_i has length smaller than x

∴ All steps between each list in a_i to x is smaller than x to a_i in O

∴ total_step (A) < total_step (O) (O is not optimal)

∴ There doesn't exist any difference

∴ A is the optimal solution

Any algorithm based on this idea should get most of the available marks. Some students may decide to sort the entire set of lists in each iteration. This is unnecessary and should cost a couple of marks. Another predictable error is to start by sorting the lists by length, but then merge all the original lists in pairs without including the merged lists until all the original lists have been merged at least once. That's a bigger error since it can easily result in a non-optimal merge order.

An answer that employs the greedy principle but is based on an incorrect sort criterion (for example, sorting the lists in descending size order) should get at least 5 out of 10, just for understanding the principle.

b) [10 marks] Outline the structure of a proof of correctness for your algorithm, describing what you would do at each stage of the proof. You are not required to fill in the details of the proof (but feel free to do so if you wish!)

My solution would look something like this:

1. **Prove that the algorithm finds a solution to the problem. I would argue that the algorithm repeatedly merges lists until only one list remains.**

2. **Prove that the algorithm's solution is optimal, using proof by induction:**

    2a. Establish a base case. I would argue that when there are <= 2 files, there is only one solution and the algorithm finds it

    2b. Inductive Hypothesis: assume the algorithm finds an optimal solution when there are < n lists, for some n

    2c. Prove that the algorithm's first decision (ie which files to merge first) is part of an optimal solution. I would argue that if the smallest lists are not merged first, the number of operations will not be increased if we change the merge order to make this merge first.

    2d. Prove that the algorithm's solution is optimal. I would argue that the inductive Hypothesis guarantees that the rest of the algorithm's solution is optimal, and that this combines with the optimality of the first decision to give an optimal solution to the whole problem.

Marking:

Step 1 is worth 1 mark. We often gloss over this, but it is actually important that when we talk about "the algorithm's solution" in Step 2, we are talking about a real thing.

Assuming the student uses an inductive proof, marks should be allocated as

2a. 1 mark

2b. 2 marks

2c. 3 marks – this is the most difficult step of this proof. Students can be quite vague about what they would actually do in this step, and that's ok

2d. 3 marks – again, this is probably going to be difficult. If they remember to refer to the Inductive Hypothesis, they should get the marks.

Students may use non-inductive proof techniques as well. For example, they may adapt the technique used for Kruskal's Algorithm. The merging of lists is conceptually similar to the joining of subtrees by selecting edges of least weight. This type of proof would look something like:

1. Define a "safe" sequence of merges to be a sequence that can be extended to an optimal solution.

2. Show that the first set of merges (ie. the empty set) is safe.

3. Show that for each iteration, if the set of merges made so far is safe then making the algorithm's next selected merge results in a larger safe set of merges. This is a more complex argument than the inductive one, but they don't have to give the details. As a proof structure, it is completely acceptable.

The goal of this question is to show an understanding of what a valid proof of optimality looks like, without requiring all the details to be filled in.

QUESTION 2 (10 marks)

Professor Snope's arch-rival Doctor Phibes proposes the following Greedy Algorithm for the Max Independent Set problem (recall: this problem asks for the largest possible set of vertices in a graph G such that none of them are joined by any edges):

```
1.  Sort the vertices of G into ascending degree order (ie, vertices of lowest degree
are at the beginning of the sorted list)
2.  Let S ={}
3.  For each vertex v in the sorted list:
        if S + {v} is an independent set:
            add v to S
```

a) [5 marks] Assuming that the graph G is represented by a set of adjacency lists, and that set membership can be tested in constant time, what is the complexity of Phibes' algorithm? Explain your answer.

If the degrees are not given, we can determine all vertex degrees in O(n^2) time. We can sort the vertices by degree in O(n*log n) time. The loop iterates n times, and each iteration takes O(n) time, which gives O(n^2) time for the loop. Thus the algorithm runs in O(n^2) time.

Marking: It's ok if they assume the vertex degrees are given. 2 marks should be allocated to giving the sort complexity as O(n*log n), and 3 marks should go to recognizing that the loop is O(n^2). It is also correct to say that the loop executes in O(m) time, where m is the number of edges in the graph.

b) [5 marks] Do you believe that Phibes' algorithm always finds a maximum independent set? Explain your answer.

No. The algorithm runs in O(n^2) time, and Max Independent Set is a known NP-Complete problem (technically it is NP-Hard, but we have not made that distinction in this course). If Phibes' algorithm always finds a max

QUESTION 2 (10 marks)

Professor Snope's arch-rival Doctor Phibes proposes the following Greedy Algorithm for the Max Independent Set problem (recall: this problem asks for the largest possible set of vertices in a graph G such that none of them are joined by any edges):

1. Sort the vertices of G into ascending degree order (ie, vertices of lowest degree are at the beginning of the sorted list)
2. Let S ={}
3. For each vertex v in the sorted list:

```
if S + {v} is an independent set:
    add v to S
```

a) [5 marks] Assuming that the graph G is represented by a set of adjacency lists, and that set membership can be tested in constant time, what is the complexity of Phibes' algorithm? Explain your answer.

Suppose the degree of each vertices

↳ Sort the vertices by their degree in decreasing order takes $O(n\log n)$

↳ { For each vertex takes n

  determine if $S+\{v\}$ is an independent set takes n

↳ $O(n^2)$

∴ The complexity is $O(n^2)$

(b)

independent set, then P = NP. Since this is almost certainly not true, I don't believe that the algorithm always finds a maximum independent set.

QUESTION 3 (20 marks)

You have **n** cases of maple syrup to sell to **n** customers. Let $\mathbf{s_i}$ be the number of litres of syrup in case i. Let $\mathbf{p_j}$ be the price per litre that customer j will pay. You can sell one case to each customer.

Example: suppose you have 2 cases containing 10 and 20 litres each, and 2 customers who will pay $5 per litre and $6 per litre.

If you sell the 10 litre case to Customer 1 and the 20 litre case to Customer 2, your income is 10*5 + 20*6 = 170.

However, if you sell the 10 litre case to Customer 2 and the 20 litre case to Customer 1, your income is only 10*6 + 20*5 = 160. Clearly the first solution gives you a larger income.

a) **[10 marks]** Create a Greedy Algorithm that will match cases with customers so that your total income is maximized. Express your algorithm in clear pseudo-code.

A useful fact: if $s_1 > s_2 > 0$ and $p_1 > p_2 > 0$, then

$$s_1{*}p_1 + s_2{*}p_2 \quad > \quad s_1{*}p_2 + s_2{*}p_1$$

The basic principle is to sell the largest case of syrup to the customer who will pay the most per litre. The algorithm is:

```
1. Sort the cases of syrup into descending order by size.
2. Sort the customers into descending order by price they will pay.
3. For i = 1 .. n:
       sell case i to customer i
```

QUESTION 3 (20 marks)

You have **n** cases of maple syrup to sell to **n** customers. Let $\mathbf{s_i}$ be the number of litres of syrup in case i. Let $\mathbf{p_j}$ be the price per litre that customer j will pay. You can sell one case to each customer.

Example: suppose you have 2 cases containing 10 and 20 litres each, and 2 customers who will pay $5 per litre and $6 per litre.

If you sell the 10 litre case to Customer 1 and the 20 litre case to Customer 2, your income is 10*5 + 20*6 = 170.

However, if you sell the 10 litre case to Customer 2 and the 20 litre case to Customer 1, your income is only 10*6 + 20*5 = 160. Clearly the first solution gives you a larger income.

a) **[10 marks]** Create a Greedy Algorithm that will match cases with customers so that your total income is maximized. Express your algorithm in clear pseudo-code.

```
① Sort the Customers by their price in decreasing order
② Sort the Cases by litres in decreasing
③ i = 0
④ income = 0
⑤ while i < num_Customers :
      income += Sell case [i] to Customer [i]
      i++
```

(b) first decision made is correct

Suppose algorithm sell cases to $A = \{C_1, C_2, C_3 \cdots C_k\}$

optimal solution sell cases to $O = \{O_1, O_2, O_3 \cdots O_k\}$

Suppose $C_1 \neq O_1$

Customer who paid most exist in O_i where $i > 1$ $(O_i = C_1)$

Construct $O^* = \{C_1, O_2, O_3 \cdots O_1, \cdots O_k\}$ swap C_1 and O_1

$$\text{total\_value}(O^*) - \text{total\_value}(O) = (C_1 * \text{Case}[1] - O_1 * \text{Case}[1]) + (O_1 * \text{Case}[i] - C_1 * \text{Case}[i])$$

$$= \text{Case}[1](C_1 - O_1) + \text{Case}[i](O_1 - C_1)$$

$$= \text{Case}[1](C_1 - O_1) - \text{Case}[i](C_1 - O_1)$$

$$= \underbrace{(C_1 - O_1)}_{\downarrow} \underbrace{(\text{Case}[1] - \text{Case}[i])}_{\downarrow} > 0$$

$C_1 - O_1 > 0$ $\text{case}[1] - \text{Case}[i] > 0$

$\therefore O^*$ is a better solution

Marking: as usual, if they show they understand what a greedy algorithm is (sort followed by selection) they should get at least 50%

b) [10 marks] Prove that the first decision made by your algorithm is correct.

The first decision the algorithm makes is to match the maximum s with the maximum p. Let these values be sm and pm.

Let O be any optimal solution, and let sm be matched with px in O and let pm be matched with sy in O. If px = pm, then we can "swap" px and pm without affecting the total value. This gives a new optimal solution that matches the algorithm's first choice and we are done.

Similarly, if sy = sm, we can swap sy and sm without reducing the value, giving a new optimal solution that matches the algorithm's first choice.

The remaining possibility is that px != pm and sx != pm. Since sm and pm are the maxima in their respective sets, we know px < pm and sx < pm.

Applying the useful fact, we see that sm*pm + sy*px > sm*px + sy*pm, so "swapping" px and pm in O would increase the value … but that isn't possible because O is optimal. Thus it must be the case that px = pm or sy = sm and we can modify O to match the algorithm's first decision without affecting the value.

Marking: This is the most difficult question on the test – marking can be quite generous. If they show that they understand the concept of taking an optimal solution and showing that it either contains the algorithm's first choice or, if it doesn't, it can be modified into another optimal solution that does match the algorithm's first choice, they should get most of the marks for this question.

CISC/CMPE-365*
Test #2
October 22, 2015

Student Number (Required) ______________________

Name (Optional)________________________________

This is a closed book test. You may not refer to any resources.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, but will not be reconsidered after the test papers have been returned.

The test will be marked out of 50.

Question 1	/30
Question 2	/20
TOTAL	/50

The point is, ladies and gentleman, greed is good. Greed works, greed is right. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. Greed in all its forms, greed for life, money, love, knowledge has marked the upward surge in mankind.

— Michael Douglas as Gordon Gekko,
Wall Street
(1987)

General marking philosophy: a student who gives enough of an answer to show they understood what they were supposed to do, even if they couldn't do it (or made lots of errors while doing it) should get at least 50% on that question.

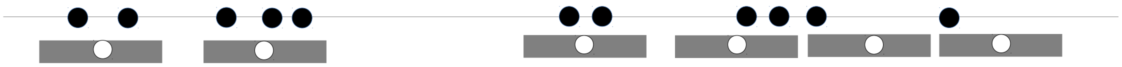
Full marks should be given if a solution is sound and not missing anything important.

Feel free to give marks like 14.5/15 to a solution that is correct but contains a minor error.

A student should only get 0 on a question if they made no attempt to answer it at all.

Question 1 (30 Marks)

You have won the contract to install Wi-Fi nodes along a very straight and sparsely populated stretch of road which runs due east and west across the tiny nation of Occiput. There are N houses along the road – each house is identified by its distance from the east end of the road. Each house is located right on the road, not set back from the road. Your assignment is to install Wi-Fi nodes along the road so that each house is no more than 1 kilometre from a node. You can install nodes anywhere along the road – the nodes do not have to be located at houses. You want to install **as few nodes as possible**.



This figure illustrates an instance of the problem and one possible solution. The black dots represent houses, the white dots represent Wi-Fi nodes, and the grey bars show the "1 km in each direction" range of each Wi-Fi node. **The solution shown is not optimal.**

a) (**10 marks)** Give a Greedy Algorithm to find an optimal (minimal) set of locations for the Wi-Fi nodes. (Hint: consider the west-most house – how far east of that house can you place the first node?)

```
sort the houses in west-to-east order
while at least one house is not covered:
    let x be the west-most uncovered house
    place a Wi-Fi node exactly 1 km east of house x
```

or equivalently:

```
sort the houses in west-to-east order
for h in the sorted list of houses:
    if h is not covered by a previously placed Wi-Fi node:
        place a Wi-Fi node exactly 1 km east of h
```

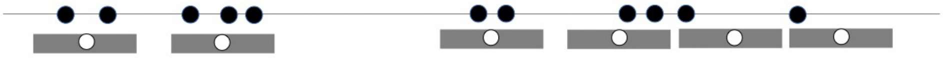
Marking: the algorithm can be run from east-to-west without affecting its correctness.

Deduct 1 mark if the student forgets to sort the houses.

For algorithms that are greedy but do not find an optimal solution (for example, "place the first node where it can cover the most houses") give about 7 marks. For algorithms that aren't really greedy (for example "place a Wi-Fi node right on every house") give about 5

Question 1 (30 Marks)

You have won the contract to install Wi-Fi nodes along a very straight and sparsely populated stretch of road which runs due east and west across the tiny nation of Occiput. There are N houses along the road – each house is identified by its distance from the east end of the road. Each house is located right on the road, not set back from the road. Your assignment is to install Wi-Fi nodes along the road so that each house is no more than 1 kilometre from a node. You can install nodes anywhere along the road – the nodes do not have to be located at houses. You want to install **as few nodes as possible**.



This figure illustrates an instance of the problem and one possible solution. The black dots represent houses, the white dots represent Wi-Fi nodes, and the grey bars show the "1 km in each direction" range of each Wi-Fi node. **The solution shown is not optimal.**

a) **(10 marks)** Give a Greedy Algorithm to find an optimal (minimal) set of locations for the Wi-Fi nodes. (Hint: consider the west-most house – how far east of that house can you place the first node?)

```
1、Sort the houses by their distance to east in decreasing
2、while house-cover != 0 :
3、    place the wifi at 1Km east to the house
4、    update house-cover
```

(b) Suppose $\begin{cases} \text{Algorithm finds wifi} : \{a_1, a_2, a_3 \dots a_k\} \\ \text{Optimal solution is} : \{o_1, o_2, o_3 \dots o_k\} \end{cases}$

Suppose $a_1 \neq o_1$

by the design of the algorithm, a_1 will be 1km east of the west most house, therefore the distance between the westmoost house and o_1 has to be less than 1km (or else west most house will be ignored),

$\therefore$ a_1 is closer to o_2, which means a_1 can cover every house between o_1 and o_2

$\therefore$ Replace a_1 into the optimal solution $= \{a_1, o_2, o_3 \dots o_k\}$ is a feasible solution

note that $|O^*| = |O|$

Therefore a_1 is part of an optimal solution

(c) Base case: For one house, algorithm put only one wifi

Induction Hypothesis: For $n \leq k$ houses, algorithm finds the best solution

k+1 : Prove a_1 is part of an optimal

Consider a reduced problem that westmost house is eliminated, we only need to consider k house. By IH: algorithm finds the optimal

Therefore, by adding one house to the reduced problem, only one wifi router will be added $|A|+1 = |O|+1 \Rightarrow A$ is optimal

marks. For algorithms that do not find a feasible solution, give about 4 marks.

b) **(10 marks)** Prove that the first choice your algorithm makes for a node location is correct (i.e. that there is an optimal solution that contains this location as its first location).

The algorithm's first choice is to place a node 1 km east of the west-most house. Call this location a1.

Let O be an optimal solution, and let o1 be the west-most node in O. o1 cannot be east of a1, since then the west-most house would not be covered by any node in O. Thus either o1 = a1, or o1 is west of a1. If o1 = a1 then a1 is contained in an optimal solution. If o1 is west of a1, then a node at a1 will cover all the houses that a node at o1 covers. Thus we can remove o1 from O and replace it with a1. This gives a feasible solution with the same cardinality as O, ie an optimal solution that contains a1.

Thus there is an optimal solution that contains a1.

Marking: The key idea here is that the algorithm's first choice can be substituted into any optimal solution that doesn't already contain it. If the student has that idea, they should get at least 6 marks, even if they couldn't come up with a proof.

c) **(10 marks)** Complete the proof that your algorithm finds an optimal solution.

Clearly if there is only 1 house, any optimal solution contains one node. The algorithm finds an optimal solution in this base case.

Assume the algorithm finds an optimal solution when there are <= n houses.

Suppose there are n+1 houses. Let A = {a1, a2, … as} be the algorithm's solution, and let O = {a1, o2, o3, …, ot} be an optimal solution, in west-to-east order, containing a1 (we know that such a solution exists). We need to show |A| = |O|

By our inductive assumption, {a2, …, as} is an optimal solution to the problem of covering all the houses not covered by a1. But this is exactly the same problem that is solved by {02, …, ot}. Therefore |{a2 …. as}| <= |{o2 …. ot}|. Therefore |A| <= |O|. |A| < |O| is impossible since O is optimal. Therefore |A| = |O|, so A is optimal too.

Marking part c): Induction is a very natural way to prove this. The base case is worth 3 marks, and the inductive part is worth 7. If they have the basic idea of induction but don't give a sound proof, they should still get at least 6 marks.

An alternative, non-inductive proof might look like this:

Let A = {a1, a2, …., as} be the algorithm's solution, and let O = {o1, o2, …., ot} be an optimal solution. Using the argument already given, we can see that O' = {a1, o2, …., ot} is also an optimal solution. Now we can make a similar argument that a2 can be used to replace o2, giving O'' = {a1, a2, o3, …., ot} is a feasible solution with the same cardinality as O, so O'' is also optimal. Repeating this argument, we replace all the o's with a's, always maintaining optimality. We end up with A being optimal.

Question 2 (20 Marks)

You have landed a prestigious new job, hiring guards for the National Prison for Disgraced Politicians (a very crowded place). The prisoners must be guarded from 6 AM to 6 PM. There are a total of **n** guards, but each guard is only available for a specific time period during the day: Guard G_i will work during the interval $[s_i, f_i]$, where $0 <= s_i < f_i <= 24$. Each guard is payed the same amount, regardless of how long their shift is. Since you are paying them out of your own salary, your goal is to hire as few guards as possible.

You may assume that there is a feasible solution – there are enough guards to cover the whole day.

(a) **(10 marks)** Give a Greedy Algorithm to find an optimal solution (i.e. minimal number of guards) subject to the constraint that there must be at least one guard on duty at all times between 6 AM and 6 PM. The total time period covered may start before 6 AM and may end after 6 PM.

In pseudo-code:

```
Sort the guards by their start times (earliest first)
Time_covered = S-1
index = 1
while Time_covered < F:
    best_guard = nil
    best_guard_end = 0
    while $s_{index}$ <= Time_Covered + 1:
        if $f_{index}$ > best_guard_end:
            best_guard = index
            best_guard_end = $f_{index}$
        index ++
    hire guard $G_{best\_guard}$    # ie add $G_{best\_guard}$ to the solution
    Time_covered = best_guard_end
```

in English:

Sort the guards by their start times (earliest first).

From the guards that cover S, choose the one with the latest finish time. Continue with that time + 1 as the new start time.

Question 2 (20 Marks)

You have landed a prestigious new job, hiring guards for the National Prison for Disgraced Politicians (a very crowded place). The prisoners must be guarded from 6 AM to 6 PM. There are a total of **n** guards, but each guard is only available for a specific time period during the day: Guard G_i will work during the interval $[s_i, f_i]$, where $0 <= s_i < f_i <= 24$. Each guard is payed the same amount, regardless of how long their shift is. Since you are paying them out of your own salary, your goal is to hire as few guards as possible.

You may assume that there is a feasible solution – there are enough guards to cover the whole day.

(a) **(10 marks)** Give a Greedy Algorithm to find an optimal solution (i.e. minimal number of guards) subject to the constraint that there must be at least one guard on duty at all times between 6 AM and 6 PM. The total time period covered may start before 6 AM and may end after 6 PM.

```
(a) Sort the guards base on (fi - Si) in decreasing order
    time = 6AM
    while time < 6PM :
            For i in len ( guard) :
                if guard[i].Si <= time :
                    break;
                i++
        remove guard[i] from guard_list
        schedule guard[i] to work
        time = guard[i].fi
# 只用 (fi - Si) 排序的话会带来问题: what if si = 3AM, fi = 6AM    X
```

```
↳ (a) Sort the guards base on fi in decreasing order
    time = 6AM
    while time < 6PM :
            For i in len ( guard) :
                if guard[i].Si <= time :

                    break;
                i++
        remove guard[i] from guard_list
        schedule guard[i] to work
        time = guard[i].fi
      排序的话会带来问题: what if si - 3AM, fi = 6AM
```

(b) **(10 marks)** Explain why your algorithm would not work if there is an added constraint that each guard has a first name (Kim, Pat, Kelly, etc) and you cannot hire two guards with the same first name.

We never have to go back and look at a guard twice because we only reject a guard if we have found a better one (ie one who covers the same required start time and whose end-time is later).

Marking: Similar to Question 1 (a). The algorithm can be presented descriptively or in code or pseudo-code.

Students are not required to give any justification for their algorithm. I included the "never have to go back" comment for the benefit of the reader.

Note: students may have interpreted the question to mean that when guards relieve each other, they must overlap (eg if the first guard ends her shift at time x, then the second must start no later than time x-1). This is a reasonable interpretation and should not be penalized. It doesn't affect the structure of the algorithm, just the criterion for deciding if a guard can feasibly be added to the solution.

(b) **(10 marks)** Explain why your algorithm would not work if there is an added constraint that each guard has a first name (Kim, Pat, Kelly, etc) and you cannot hire two guards with the same first name.

Suppose there are two guards named Kim – call them Kim1 and Kim2. The algorithm's first choice might be Kim1, and then on a later iteration, the best – or perhaps the only – choice might be Kim2. If the algorithm chooses Kim2, it violates the constraint – if it doesn't choose Kim2, it may not find a solution at all.

Thus we can construct an instance where the algorithm fails.

Marking: the key idea is that for a greedy algorithm to be successful, its choices should be based on purely "local" information. It should not be the case that the optimal first choice needs to consider future optimal – or essential – choices.

Students can explain this clearly or give an example for full marks.

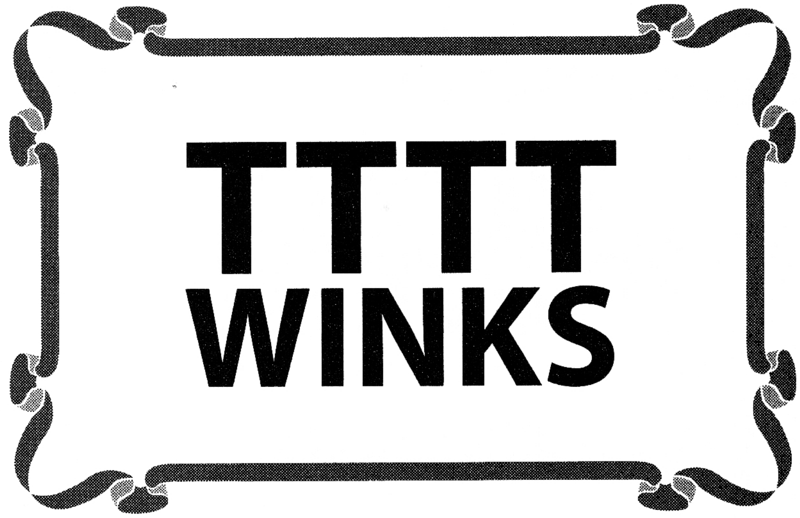
An alternative (and fully acceptable) demonstration of the failure would be to show that proof by induction would not be possible. We can assume that the algorithm makes an optimal first choice and that it finds an optimal solution to the reduced problem after the first guard is chosen, but that does not guarantee that the algorithm's first choice and the optimal solution to the reduced problem can be combined – as described above, the constraint on names might be violated.

Give part marks in the range 7 to 10 for answers that come close to giving a good explanation or an example of how the algorithm could fail.

Give marks of 5 or less for answers that show some understanding but cannot identify (in any clear way) how the algorithm might fail.

Bonus Question (0 marks):

What is the meaning of this figure?



CISC/CMPE-365*
Test #2
October 21, 2016

Student Number (Required) ______________________

Name (Optional)________________________________

This is a closed book test. You may not refer to any resources.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, but will not be reconsidered after the test papers have been returned.

The test will be marked out of 50.

Question 1	/12
Question 2	/25
Question 3	/12
Question 4	/1
TOTAL	/50

General marking philosophy: a student who gives enough of an answer to show they understood what they were supposed to do, even if they couldn't do it (or made lots of errors while doing it) should get at least 50% on that question.

Full marks should be given if a solution is sound and not missing anything important.

Feel free to give marks like 9.5/10 to a solution that is correct but contains a minor error.

Students may come up with solutions that are completely different from mine but still completely correct. Correct solutions should get full marks even if they don't match mine.

Students should always get a few marks for trying a question. The only way to get a 0 is to leave the page blank or write something completely irrelevant.

Question 1 (12 Marks)

(a) **[6 marks]** Show the bitstring codes that result from applying the Huffman Coding algorithm to a string containing the following set of letters with the indicated frequencies:

a	b	c	d	e	f	g	h
1	1	2	3	5	8	13	21

Solution: most students will probably draw the tree to explain how they get the final bitstrings. The tree will probably look something like this

```
a     b     c     d     e     f     g     h
 \   /     /     /     /     /     /     /
  ab      /     /     /     /     /     /
     \   /     /     /     /     /     /
      abc     /     /     /     /     /
        \   /     /     /     /     /
         abcd    /     /     /     /
           \    /     /     /     /
            abcde    /     /     /
              \     /     /     /
               abcdef    /     /
                  \     /     /
                   abcdefg   /
                       \    /
                       abcdefgh
```

with the edges labeled "0" and "1" . It is not necessary to label the internal vertices. If the "up and left" edges are "0" and the "up and right" edges are "1" this gives

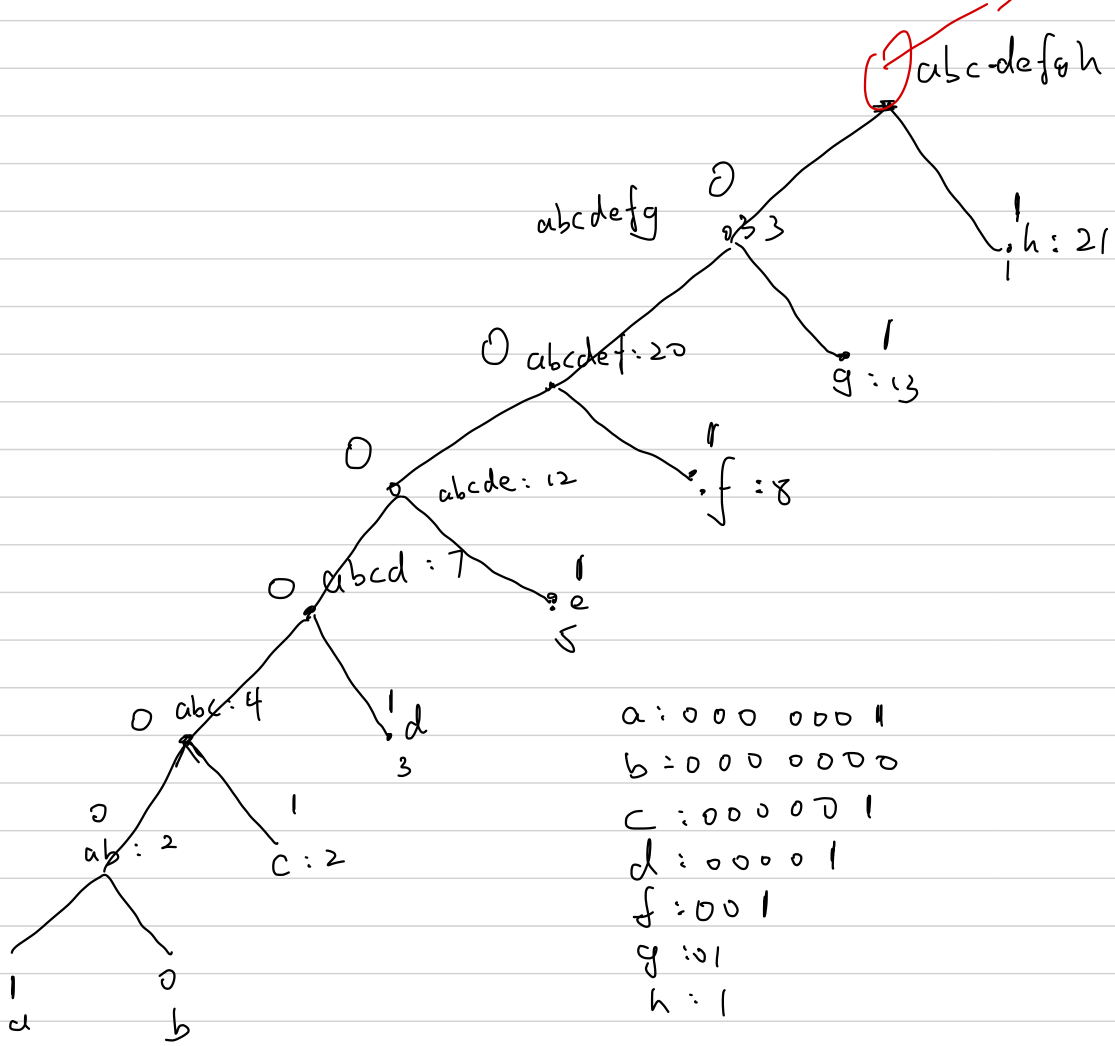
a: 0000000 b: 0000001 c: 000001 d: 00001 e: 0001 f: 001 g: 01 h: 1

Question 1 (12 Marks)

(a) **[6 marks]** Show the bitstring codes that result from applying the Huffman Coding algorithm to a string containing the following set of letters with the indicated frequencies:

a	b	c	d	e	f	g	h
1	1	2	3	5	8	13	21

不给数字!!!



a : 000 0001
b : 000 0000
c : 000001
d : 00001
f : 001
g : 01
h : 1

(b) **[3 marks]** Is your answer in (a) unique? Why or why not?

No, it is not unique. Exchanging the "0" and "1" labels on any pair of edges that both go "up" from a single vertex with give an equivalent code. It is also true that swapping all "0" and "1" edge-labels will give an equivalent code.

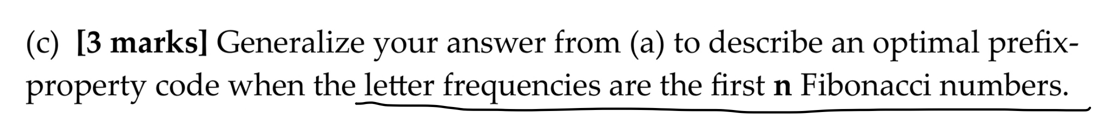
(c) **[3 marks]** Generalize your answer from (a) to describe an optimal prefix-property code when the letter frequencies are the first **n** Fibonacci numbers.

The highest frequency letter gets a bitstring of length 1. The second highest frequency letter gets a bitstring of length 2, and so on down to the last two letters, which both get bitstrings of length n-1. The bitstrings must obey the prefix-property rule.

If a student does not know the definition of the Fibonacci sequence ...if they extrapolated from part (a) in a plausible fashion and gave a decent answer, that's ok

(b) No it's not unique, 隨便互換 都能得到一樣的 table

cc)



(c) **[3 marks]** Generalize your answer from (a) to describe an optimal prefix-property code when the letter frequencies are the first **n** Fibonacci numbers.

Question 2 (25 marks)

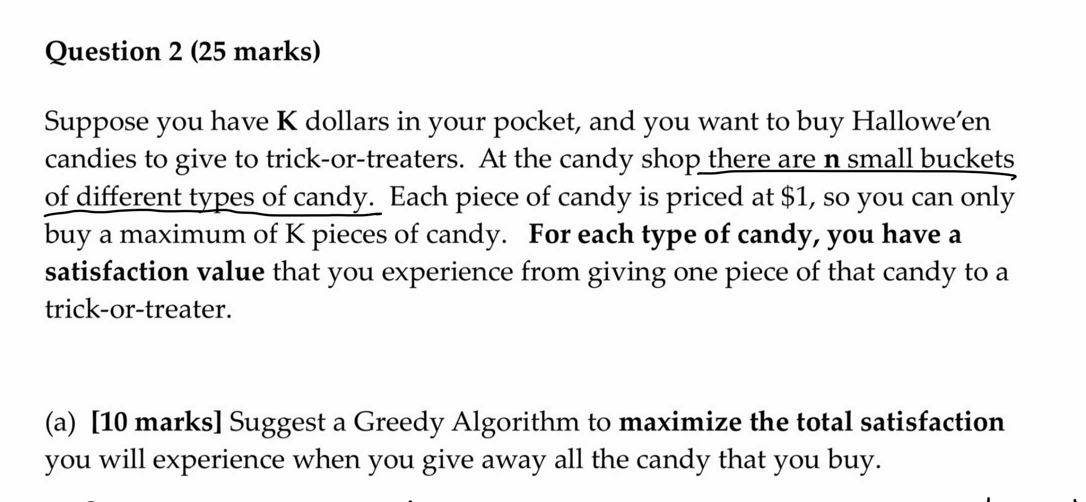
Suppose you have **K** dollars in your pocket, and you want to buy Hallowe'en candies to give to trick-or-treaters. At the candy shop there are **n** small buckets of different types of candy. Each piece of candy is priced at $1, so you can only buy a maximum of K pieces of candy. **For each type of candy, you have a satisfaction value** that you experience from giving one piece of that candy to a trick-or-treater.

(a) **[10 marks]** Suggest a Greedy Algorithm to **maximize the total satisfaction** you will experience when you give away all the candy that you buy.

```
Sort the candies in descending order by their satisfaction value.
$R = K$
while $R > 0$ and there are still some candies left to buy:
    buy a candy with the highest satisfaction value
    $R = R – 1$
```

or

```
Sort the candies in descending order by their satisfaction value.
$R = K$
while $R > 0$ and there are still some candies left to buy:
    buy as many candies as possible with the highest satisfaction value
    $R = R$ – the number of candies bought on the line above this one
```



Question 2 (25 marks)

Suppose you have **K** dollars in your pocket, and you want to buy Hallowe'en candies to give to trick-or-treaters. At the candy shop there are **n** small buckets of different types of candy. Each piece of candy is priced at $1, so you can only buy a maximum of K pieces of candy. **For each type of candy, you have a satisfaction value** that you experience from giving one piece of that candy to a trick-or-treater.

(a) **[10 marks]** Suggest a Greedy Algorithm to **maximize the total satisfaction** you will experience when you give away all the candy that you buy.

```
(a) 1、Sort the types of Candy by satisfaction value in decreasing ordor
    2、i=0
    3、while money > 0 :
    4、    buy as much a[i] as you can
    5、    if no more a[i] left : i++
    6、    deduct money
```

(b) **[5 marks]** Prove that your algorithm's first choice is optimal (i.e. that there is an optimal solution that makes the same choice)

Suppose that algorithm finds $\{a_1, a_2, a_3 \cdots a_K\}$

optimal solution $\{O_1, O_2, O_3 \cdots O_K\}$

Assume algorithm's first choice is not optimal

$\therefore a_1 \neq O_1$

Based on design of the algorithm, a_1 is definitely the candy with most Satisfactory value

$\therefore O_1$ is not the candy with most satisfactory value

let O^* be $\{a_1, O_2, O_3, \cdots O_K\}$, this solution is feasible because

$|O^*| = |O|$, no more moey cost

and total_val (O^*) - total_val (O) $= (a_1 - O_1) > 0$

$\therefore a_1$ is part of an optimal solution

(b) **[5 marks]** Prove that your algorithm's first choice is optimal (i.e. that there is an optimal solution that makes the same choice)

Consider an optimal solution that does not include as many of the highest satisfaction value candies as the Algorithm's solution. Then we can replace some equal-or-lower value candies in the optimal solution, without lowering its total value, using the left-over highest value candies. Thus there is an optimal solution that matches the number of highest-value candies in the Algorithm's solution.

(c) **[10 marks]** Complete the proof that your algorithm finds an optimal solution to the problem.

Continuing the argument above, we can start with an optimal solution that matches the Algorithm's choice with respect to the highest-value candy. Using the same reasoning, we can find an optimal solution that also matches the Algorithm's choice with respect to the second-highest-value candy, and so on. Eventually we reach an optimal solution that is identical to the Algorithm's solution ... hence the Algorithm's solution is optimal.

Proof by induction is also a reasonable approach.

Marking: students seem to have interpreted this problem in a variety of ways. For example, some students assumed the buckets are sealed and you can't pick individual candies out. This obviously changes the answer, but they can still come up with a greedy algorithm (although it won't always give the optimal solution because this interpretation makes the problem equivalent to the 01 Knapsack Problem). Other students assumed that the small buckets contain infinite numbers of candies (?) … which also affects the details of the answer, but not its principle. If they give an answer that is correct relative to their interpretation, that's ok.

(b) **[5 marks]** Prove that your algorithm's first choice is optimal (i.e. that there is an optimal solution that makes the same choice)

Suppose that algorithm finds $\{a_1, a_2, a_3 \dots a_k\}$
optimal solution $\{O_1, O_2, O_3 \dots O_k\}$

Assume algorithm's first choice is not optimal
$\therefore a_1 \neq O_1$
Based on design of the algorithm, a_1 is definitely the candy with most satisfactory value

$\therefore O_1$ is not the candy with most satisfactory value
let O^* be $\{a_1, O_2, O_3, \dots O_k\}$, this solution is feasible because
$|O^*| = |O|$, no more moey cost
and $\text{total\_val}(O^*) - \text{total\_val}(O) = (a_1 - O_1) > 0$
$\therefore a_1$ is part of an optimal solution

C. Complete your proof

↳ Base case: when only have 1 dollar, buy the most satisfactory one
Hypothesis: Assume for $n \leq K$, algo finds the optimal solution
$K+1$: Proof first solution is part of an optimal solution
Suppose the problem reduce to K dollar
$|A| = |O|$
add one dollar
both algo will choose the most satisfactory like the optimal solution

Question 3 (12 Marks)

Is Dijkstra's Algorithm for finding least-weight paths in a graph with positive edge-weights a Greedy Algorithm?   Why or why not?

Case for Yes: on each iteration, the algorithm chooses the best option available to it. It never looks forward to anticipate future choices or back to revisit previous choices. This is the essence of the Greedy strategy.

Case for No: Greedy algorithms are supposed to start with a sort. Dijkstra's Algorithm does not start with a sort … so it is not a greedy algorithm

Marking: I'm willing to accept either "Yes" or "No" for this … but they have to give a decent reason for their answer.

Question 4 (1 mark)

Consider the following Greedy Algorithm for CNF-SAT:

```
sort the boolean variables in the expression in descending order based on
    how many terms they occur in
for each boolean variable, set it to True unless its negation has already
    been set to True
```

True or false: If E is a satisfiable expression in CNF form, this algorithm will always find a truth assignment that satisfies E

true

FALSE

The correct answer is False

CMPE/CISC-365*
Test #2
October 22, 2019

Student Number (Required) ______________________

Name (Optional)________________________________

This is a closed book test. You may refer to one 8.5 x 11 data sheet.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, **but will not be re-marked under any circumstances.**

The test will be marked out of 50.

Question 1	/16
Question 2	/12
Question 3	/20
Question 4	/2
TOTAL	/50

"There is a very fine line between loving life and being greedy for it."

— Maya Angelou

QUESTION 1 (16 Marks)

Suppose we have a computer which is based on the trinary system, rather than binary. The fundamental unit of memory of such a system is called a trit (instead of bit). We represent everything with tritstrings consisting of 0's, 1's and 2's. In such a system, the standard representation of the letter "A" might be "102210", "B" might be "102211" etc.

Part A : [8 Marks]

Adapt the Huffman Coding scheme to the trinary system, and give a clear description of your modified algorithm for constructing variable length trinary codes. You are not required to prove that your algorithm produces optimal trinary codes.

Solution:

Sort the characters in the source document according to their frequency (same as the original algorithm)

Build a trinary tree as follows:
choose the three characters with the lowest frequency, add a parent that has their combined frequencies, and put a 0, a 1 and a 2 on the edges joining them to their parent.
Remove the three characters from the set and add their parent (as a new character) to the set.
Repeat until there is a single root that represents the combination of all the characters.

QUESTION 1 (16 Marks)

Suppose we have a computer which is based on the trinary system, rather than binary. The fundamental unit of memory of such a system is called a trit (instead of bit). We represent everything with tritstrings consisting of 0's, 1's and 2's. In such a system, the standard representation of the letter "A" might be "102210", "B" might be "102211" etc.

Part A : [8 Marks]

Adapt the Huffman Coding scheme to the trinary system, and give a clear description of your modified algorithm for constructing variable length trinary codes. You are not required to prove that your algorithm produces optimal trinary codes.

Part A :

```
Sort the letters by their frequencies in increasing order
while len( letter_list ) > 1 :
    choose three least frequency letter and put them on the same level of tree
    denote them as 0, 1, 2 respectively
    assign a combined letter as their parent
    put the combined letter back and remove these three letter, the frequency
    of the combined letter is the sum of frequencies of these letters
```

Marking:

Sorting the set:	2 marks
Choosing the three smallest:	2 marks
Adding 0, 1, 2 to their codestrings:	2 marks
Replacing them by a combination item with their summed frequencies:	2 marks

A student whose answer shows a good understanding of the basic Huffman algorithm should get at least 4/8, even if they make errors in translating it to the trinary version.

They are not required to present the algorithm in terms of building a tree. They can describe the process as "add 0, 1, 2 respectively to the codestrings for the characters represented by the three lowest frequency items"

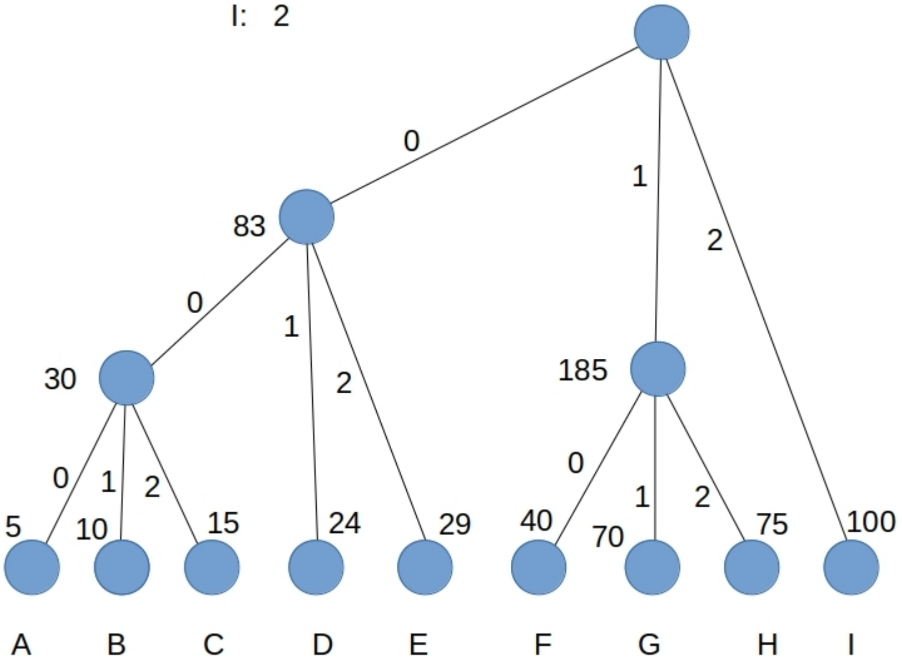
Part B : [8 marks]

Show the application of your modified algorithm to the following set of letters, where each letter is followed by its observed frequency. Show the tree and codes that your algorithm constructs:

A	5
B	10
C	15
D	24
E	29
F	40
G	70
H	75
I	100

Codestrings:

A:	000
B:	001
C:	002
D:	01
E:	02
F:	10
G:	11
H:	12
I:	2



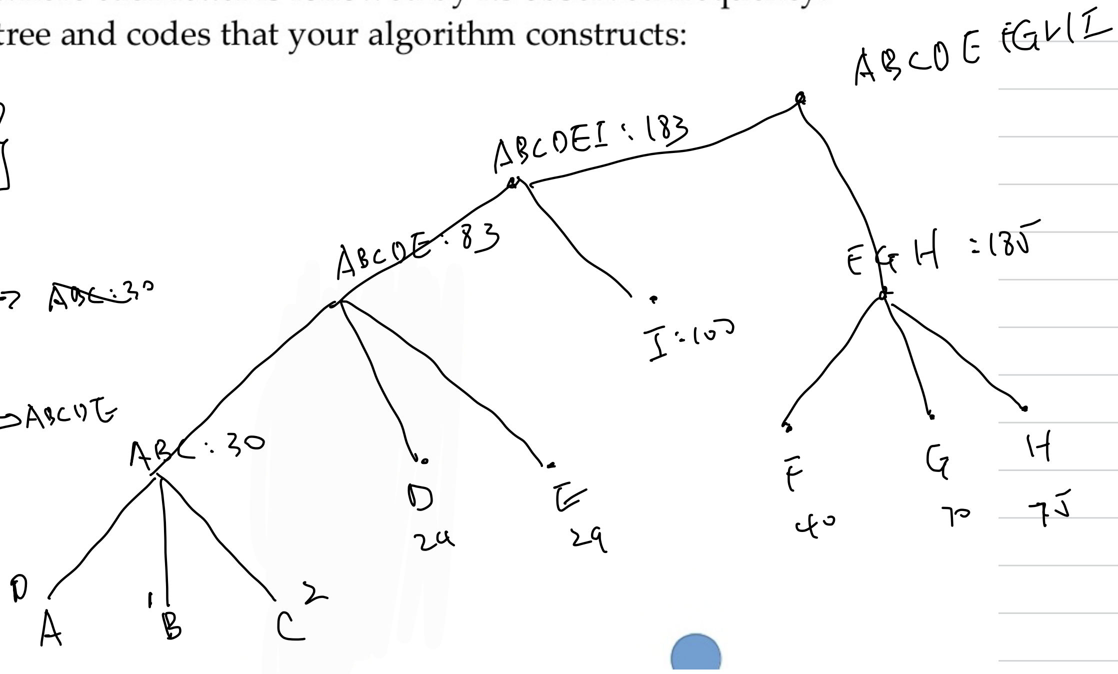
Part B : [8 marks]

Show the application of your modified algorithm to the following set of letters, where each letter is followed by its observed frequency. Show the tree and codes that your algorithm constructs:

~~A~~	5
~~B~~	10
~~C~~	15
~~D~~	24
E	29
F	40
G	70
H	75
I	100

→ ~~ABC~~:30

⟹ABCDE



Marking:

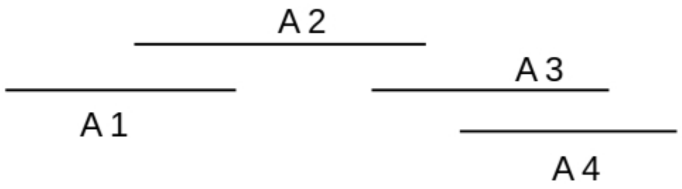
The assignment of 0,1 and 2 to the edges of the tree are arbitrary so the codestrings they construct may be very different than mine, but the lengths should be the same ("I" should have a codestring of length 1, etc.)

It is important to correctly extract the codestrings from the tree (or alternative representation). Some students may read the codestrings from the bottom up rather than from the top down, getting (for example) "100" for "B". This breaks the prefix rule and makes the code unusable.

Showing the steps of the execution:	3 marks
Showing the codestrings correctly:	5 marks
Showing the codestrings incorrectly (see explanation above):	2 marks

QUESTION 2 (12 Marks)

Suppose we have a set of n activities, each with a known start time s_i and finish time f_i . The activities may overlap. Our task is to assign the activities to rooms so that each room contains a non-overlapping subset of the activities. The goal is to use as few rooms as possible.



In this example we need two rooms: one room for A1 and A3 and the other room for A2 and A4.

A greedy algorithm for this problem: sort the activities based on start time, then assign activities to rooms. Use a new room only if the next activity overlaps with activities in all existing rooms.

```
Sort the activities based on start time and renumber them so that
                    $s_1 \le s_2 \le \cdots \le s_n$
    Room_set = {1}
    Busy_until[1] = 0
    for i = 1 to n:
        if there is any room x in Room_set with Busy_until[x] $\le s_i$ :
            assign Activity i to Room x
            Busy_until[x] = $f_i$
        else:
            add a new room to Room_set
            assign Activity i to the new room
            set Busy_until[new room] = $f_i$
```

先证明 optimal solution 就是2个房间

再证明 算法分配给房间1和房间2的活动一模一样

Question 2 continues on the next page.

Suppose the Algorithm puts Activities $1, 2, \ldots i$ into Room 1 and then puts Activity $i+1$ into Room 2.

Prove that there is an optimal solution that does exactly the same thing.

Hint: Let O be an optimal solution ...

Solution:

Let O be an optimal solution, and suppose it does something different with the first i+1 Activities. Renumber the rooms so that Activity 1 is in Room 1. This does not change the number of rooms so this renumbered solution is still optimal. Call it O′

Let Activity j+1 be the first Activity that O′ does not put in Room 1. (That is, O′ puts Activity 1, 2, ..., j in Room 1.) If j = i, then renumbering the room that contains Activity i+1 to be Room 2 exactly matches the algorithm's action. If j ≠ i then it must be true that j < i, since if j > i then the algorithm would not have put Activity i+1 into Room 2.

Let the Room containing Activity j+1 be Room k. Swap Activity j+1 and all following activities in Room k with all activities in Room 1 that follow Activity j. Because the earliest activity being swapped into Room k must have start time $\geq s_{j+1}$***, this is a feasible solution, and since it doesn't use more rooms it is also optimal.***

This new optimal solution agrees with the Algorithm's solution more than the previous one did. We can repeat this swapping action until all of Activity 1, ... Activity i are in Room 1, and Activity i+1 is in Room 2. This is exactly what the algorithm does.

Marking:

The main thing to look for here is whether the student understands how to approach this type of problem. The details are less critical.

Recognizing that our goal is to take an arbitrary optimal solution and manipulate it to create another one that matches the algorithm's choices: 4 marks

Recognizing that no optimal solution can put Activities 1, 2, ..., i+1 into the same room: 4 marks

Recognizing that we can swap Activities (or groups of Activities) between rooms without creating time-conflicts: 4 marks

Please give part marks to answers that show partial success with these aspects of the proof.

If a student takes a completely different approach and you are not sure how to grade it, please contact me.

QUESTION 3 (20 marks)

Let $S = \{s_1, s_2, ..., s_n\}$ be a set of n positive integers – possibly containing duplicates. Let k be a positive integer.

Problem: Find a **maximum-size** subset A of S that has sum $\leq$ k

For example, let S = {7, 4, 12, 1, 3, 18, 1, 240, 10} and k = 19

The solution is A = {7, 4, 1, 3, 1} (in any order) which has size 5.

Part A : [6 marks]

Create a Greedy Algorithm to solve this problem. State your algorithm in clear pseudo-code.

Solution:

Sort the values into ascending order, so $s_1 \leq s_2 \leq \ldots s_n$

```
total = 0
i = 1
solution = {}              # empty set
while total + $s_i$ <= k:
    total = total + $s_i$
    i = i+1
    solution.append($s_i$)            # or "add $s_i$ to the solution"
```

QUESTION 3 (20 marks)

Let $S = \{s_1, s_2, ..., s_n\}$ be a set of n positive integers – possibly containing duplicates. Let k be a positive integer.

Problem: Find a **maximum-size** subset A of S that has sum ≤ k

For example, let S = {7, 4, 12, 1, 3, 18, 1, 240, 10} and k = 19

The solution is A = {7, 4, 1, 3, 1} (in any order) which has size 5.

Part A : [6 marks]

Create a Greedy Algorithm to solve this problem. State your algorithm in clear pseudo-code.

```
Sort the integer in increasing order
R = {}
Sum = 0
i = 0
while Sum + S[i] ≤ k :
        Sum += S[i]
        R. append (S[i])
```

Proof : Base case : . . . .

Induction Hypothesis : . . . .

K+1 : Algorithm : $\{a_1, a_2, a_3, a_4, \ldots\}$

optimal : $\{O_1, O_2, O_3, O_4 \ldots\}$

a_1 is definitely the smallest one

suppose $a_1 \neq O_1$, O_1 is not the smallest

$\therefore a_1 < O_1$

$\therefore$ Suppose construct O^* that $\{a_1, O_2, O_3, \ldots\}$

Sum $(O^*) \leq k+1$ because $a_1 < O_1$

$\therefore O^*$ is a feasible solution

$\therefore a_1$ is part of an optimal solution

2∵ reduce problem . . . .

Marking:

Sort:	2 marks
Loop:	4 marks

No penalty if they forget to initialize the solution be empty – it's an important implementation detail but not an essential conceptual part of the algorithm.

If a student gives an incorrect algorithm, but remembered that Greedy Algorithms always sort the set then iterate through the sorted list, they should get 4/6

Part B : [14 marks]

Prove that your algorithm finds an optimal solution. Use any valid proof technique.

Solution:

Let A be the algorithm's solution and let O be any optimal solution. Sort O into ascending order.

If A and O are identical, then A is optimal.

Suppose A and O are equal up to and including s_i, but differ in the next position. The algorithm fills the next position with s_{i+1}, so O must fill the next position with s_x where x > i+1. This implies $s_x \geq s_{i+1}$, so we can remove s_x from O and replace it with s_{i+1} without pushing the total over k. This new solution has the same cardinality as O, so it is also optimal, and it has fewer differences from A.

We can repeat this sequence until we arrive at an optimal solution that has 0 differences from A – so A is optimal.

TL;DNR version of this proof:

Let O be any optimal solution that does not contain the smallest value in the set. Swap the smallest value for any value in O. The result is still optimal. Continue until all the smallest values have been swapped in. This matches the algorithm's solution.

Alternative Proof: Induction on the size of the set of values.

Base case: If |S| = 0, then the empty set is the only solution (and thus it is the optimal solution.

Inductive Hypothesis: Assume the algorithm always finds an optimal solution when the size of the set is $\leq$ n, for some n $\geq$ 0.

Let |S| = n+1, and assume the set has been sorted into ascending order. If s_1 > k, there is no nonempty subset that sums to $\leq$ k, and the algorithm correctly solves this case.

Assuming there is a non-empty solution, let A be the algorithm's solution and let O be any optimal solution that does not contain s_1. Replace any element of O with s_1. The result is still an optimal solution (call it O′), so the algorithm's first action is correct. This reduces the problem to a set of size n with a target value of $k - s_1$. By the inductive hypothesis, the algorithm finds an optimal solution to this reduced problem.

O′ also contains a solution to this same subproblem. This implies |A| = |O′| so A is optimal.

Marking:

The marking method here should be similar to Question 2, but it will depend on the proof type chosen by the student.

For the “eliminate differences” approach the essential concept is summarized in the TL;DNR version. If they express this idea clearly they should get at least 10/14

Example of an answer which is insufficiently clear:

“We should never take a larger value when a smaller one is available”. I would grade this at 7/14. The idea is there but it is not fully developed.

For the inductive approach, use this grading scheme

Base case:	4 marks
Inductive Hypothesis:	3 marks
Inductive Step:	7 marks

In each part, give partial marks for proofs that have the right ideas but don’t express them clearly.

Note that the base case can be set up with sets of size 1 rather than with the empty set.

QUESTION 4 (2 Marks)

True or false:

David Huffman was a pioneer in the field of mathematical origami.

TRUE FALSE

Solution: True

Marking:

True	**2 marks**
False	**2 marks**
No answer	**2 marks**

Yes, everyone gets 2 marks for this question. Apparently some people think I am trying to trick them with the different font sizes for TRUE and FALSE.

Dynamic Programming

↓和 greedy algorithm 的区别

* 都是把问题分成 Subproblem，不过不同的是，dynamic programming applies when the subproblem overlap, that is, when subproblems share subproblems.

* Dynamic programming algorithm solves each subsubproblem just once and then saves its answer in a table ⟶ avoid recomputing

* 一般分成 4 个步骤去设计 dynamic programming algorithm:

  1、characterize the structure for an optimal solution
  2、Recursively define the value of an optimal solution
  3、Compute the value of an optimal solution, typically in a bottom-up fashion
  4、Construct an optimal solution from computed information

∴ Dynamic Programming vs Divide and conquer

DnC 会不停地计算 subproblem (即使是一样的，重复的 subproblem)

Dp 则力求把每个 subproblem 的解给记录下来，避免重复的 subproblem

问题1: Coin change => {1, 4, 9} (用 greedy 找不到比较好的解)

↳ 用 dynamic programming !!!

Base Case: Min_Coins(0) = 0 // 没有硬币的时候不选

Min_Coins(x) = ∞ if x<0 // ∞ 确保了用 min 去选的话不会选到这个，

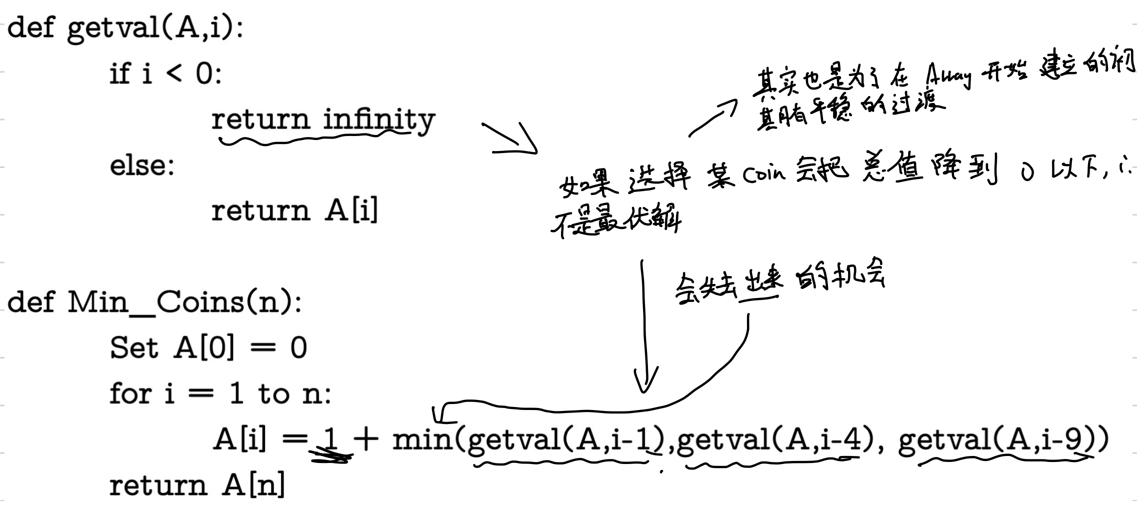
Recursive part is:

当下的这一个硬币

找出最优的 subsolution

$$\underset{\smile}{\text{Min\_Coins}(n)} = 1 + \{ \min(\text{Min\_Coins}(n-c_1), \text{Min\_Coins}(n-c_2), \ldots, \text{Min\_Coins}(n-c_k)) \}$$

因为总有值会 revisit 到过的值，所以刚刚已经记录下了到过的值，现在可以直接用计算过的值



```
def getval(A,i):
    if i < 0:
        return infinity
    else:
        return A[i]

def Min_Coins(n):
    Set A[0] = 0
    for i = 1 to n:
        A[i] = 1 + min(getval(A,i-1),getval(A,i-4), getval(A,i-9))
    return A[n]
```

如果选择某 coin 会把总值降到 0 以下, i.e 不是最优解

其实也是为了在 Array 开始建立的初期有平稳的过渡

会失去出来的机会

第4步：回溯

0	1	2	△3	4	5	6	7	△8	9	10	△11	12
0	1	2	3	1	2	3	4	2	1	2	3	③

↓ 知道了对于 $n=12$，best solution is 3 之后，要如何知道选了哪三个硬币

△代表了这是仅有的可以到达 12 的值，然后可以对比它们下面的硬币的数量，从它们到12只增加了一个硬币，所以 11 和 3 下面已经有三个硬币了，所以必不是它们，只能是 8，且知道 8 → 12 用了一枚 4 的 coin，∴ 知道了一个选择 4

0	1	2	3	△4	5	6	△7	8	9	10	11	12
0	1	2	3	1	2	3	4	②	1	2	3	3

然后看 8，是一样的做法 . . .

选择下来，可知 optimal solution 用了 3 个 4 coin

Complexty : Construct A[ ] 用了 $O(n)$
Revisit 用了 $O(n)$ } $O(n)$

问题二：The rod-cutting problem

* Given a rod of length n, and a table of prices p_i for $i = 1, 2, 3, \cdots, n$ (即不同长度的 rod 可以卖出的价格不一样)，determine the maximum revenue r_n obtained by cutting up the rod and selling the pieces.

↳ rod 的长度是整数

↳ length	1	2	3	4	5	6	7	8	9	10
price	1	5	8	9	10	17	17	20	24	30

∴ 如果 $n = 4$，最大的 r_n 必然是 2段 2 ⇒ $2 * 5 = 10$

如果 $n = 10$，最大的 r_n 即是什么都不切 ⇒ 30

所以这个题目跟上个题目是有区别的：上一个题目中，有且仅有三个选择 (1, 4, 9)，但这次不一样，given n，总是可以从 n 个选择下手：1, 2, 3, ... n

∴ Recursive part :

Base case : $\text{CutRod}(0) = 0$ # 没有木材的话，revenue 是 0

$$\text{CutRod}(n) = \max\{\, \underset{\sim}{p_1} + \text{CutRod}\underset{\sim}{(n-1)},\ p_2 + \text{CutRod}(n-2),\ \ldots,\ p_n + \text{CutRod}(n-n) \,\}$$

只切1段，首先可以得到 p_1

同时把问题压缩成了 $n-1$

```
CutRod(p, n) :          # compute the maximum
    r[0] = 0
    for j=1 to n :                  # 所以这个东西遍历了2遍，每一遍都会把当前n的能走的路径给找出来
        current_max = -∞            # 并且对比，值得注意的是：从小到大build起来时，很多值都会被记录
        for i = 1 to j :
            current_max = max (current_max, p[i] + r[j-i])
        r. append (current_max)
    return r[n]
```

O (n²)
两个loop

Reconstruct the solution (顺便加上了什么时候切第一刀) ** → 加上 S[j]是为了 reconstruct

```
Cut Rod (p, n) :        → 对于不同长度的rod，什么时候割第一刀不一样
    r[0] = 0            # maximum revenue for n
    S[0] = 0            # the optimal first cut for n
    for j = 1 to n :
        S. append (0)
        current_max = -∞
        for i = 1 to j :
            if current_max < p[i] + r[j-i] :
                current_max = p[i] + r[j-i]
                S[j] = i        # 更新第一刀的位置
        r. append (current_max)
```

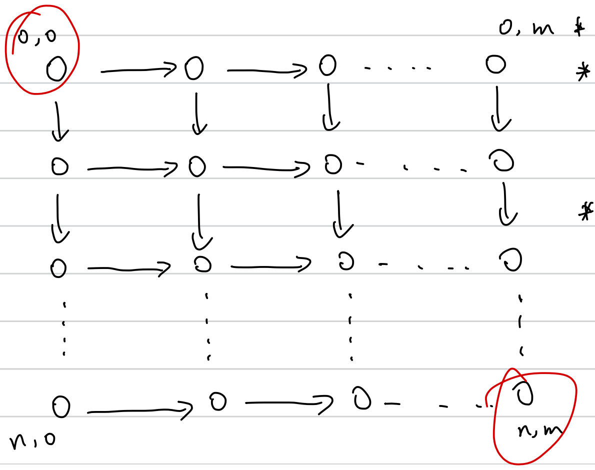
依然是通过遍历将 n 的 n种路径全部探索一遍

肯定是 O(n²)，因为有两个loop，但把subproblem都记录在一个list里面就避免了重复计算subproblem的必要，节省了不少时间，但同时也浪费了不少的空间.

```
Print_cut_Rod_solution (p, n):
(r, S) = Cut-Rod( p, n)
while  n > 0 :
    print S[n]
    n = n - P[n]
```

问题三: Shortest Path

假设目前存在 graph:



* 且每个vertex (x, y)都有edge指向 (x, y+1) 和 (x+1, y)

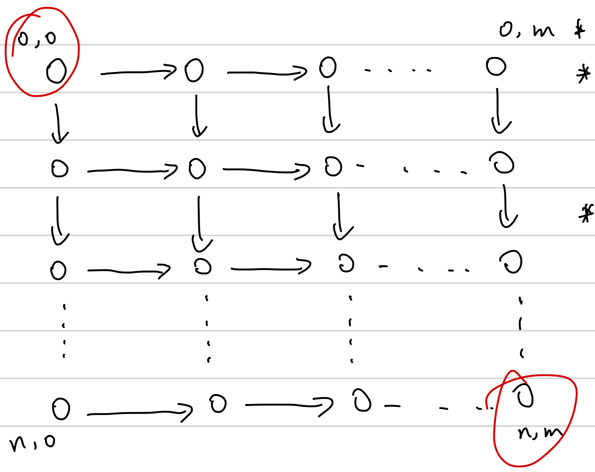
* 每条edge都对应着一个 cost :

w (a, b: c, d) : the cost of the edge from (a, b) to (c, d)

* Goal : least-cost path from (0, 0) → (n, m)

问题三：Shortest Path

假设目前存在 graph:



* 且每个vertex (x, y) 都有 edge 指向 (x, y+1) 和 (x+1, y)

* 每条 edge 都对应着一个 cost：

w(a, b : c, d) : the cost of the edge from (a, b) to (c, d)

* Goal : least-cost path from (0, 0) → (n, m)

总共有多少 Path ?

* Every path from (0, 0) → (n, m) 包含 n个 down edge 和 m个 right edge

* The total number of different paths from (0, 0) → (n, m)

= number of ways of interleaving the down and right edge

* 假设当下 n = m，那么 number of different path is $\binom{2n}{n}$?

which is $\frac{(2n)!}{(n!\,n!)} = \frac{2n)(2n-1)\cdots(n+1)}{n(n-1)(n-2)\cdots 1} = \frac{2n}{n} * \frac{2n-1}{n-1} * \frac{2n-2}{n-2} * \cdots \frac{n+1}{1}$

∴可以得知 Each term is ≥ 2, whole thing is $\geq 2^n$

∴使用 brute force 不实际

通过观察得知，从 (0, 0) → (n, m)，最后一步，要么是 (n-1, m)，要么是 (n, m-1)

Recursive version:

找寻从 0,0 → n-1, m 的 cost　加上 n-1, m → n, m 的 cost

再进行对比得到当下更好的解

$$\text{Min\_ Cost} = \min \{ \text{min\_ cost}(n-1, m) + w(n-1, m : n, m),$$
$$\text{min-Cost}(n, m-1) + w(n, m-1 : n, m) \}$$

Min_Cost(0,0) = 0

多个两个base case的原因是，当回溯到第0行或者是第0列的时候，他就只能沿着第0行；第0列回溯，而不能进行两个方向的回溯

Min_Cost(i,0) = Min_Cost(i-1,0) + w(i-1,0 : i,0)	for i > 0
Min_Cost(0,j) = Min_Cost(0,j-1) + w(0,j-1 : 0,j)	for j > 0

Min_Cost(i,j) = min { Min_Cost(i-1,j) + w(i-1,j : i,j),
Min_Cost(i,j-1) + w(i,j-1 : i,j)
}
	for i,j > 0

Subsolution的结果可以储存在一个 2-d array 中: MC [n] [m] = Min-cost (i, j)
然后遍历地去为这个 list 增添内容 (可以一行一行或一列一列地来), complexity 都是 $O(n*m)$

4. 回溯

① 可以在 MC [n] [m] 这个信息表中增添一些用来回溯的信息, 比如 MC [i] [j] 中加一个 "H", 那么就可以得知来[i] [j] 经过 [i] [j-1] / "V" → 从 [i-1] [j] 来的

② 或者是直接遍历去寻找上一个:

```
if MC[n][m] == MC[n][m-1] + w(n, m-1 : n, m):
        找 vertex = (n, m-1)                        } O(n+m)
else:
        找 vertex = (n-1, m)
```

Total Complexity : $O(n*m) + O(n+m) = O(n*m)$

问题 4: Longest Common Subsequence

↳ Subsequence: T is a subsequence of S if we can get T by deleting some characters of S.

longest common subsequence: Given two strings $P = p_1 p_2 \cdots p_n$ and $Q = q_1 q_2 \cdots q_m$ with length n, m respectively, how can we find the longest common subsequence of P and Q?

根据后面的 proof, 不难得出以下的 recurrence relation

if $p_n == q_m$:   # P和 Q 最后一个字母 相等

$$LCSL(P, Q) = 1 + LCSL(P[1 \cdots n-1], Q[1 \cdots m-1])$$

else :

$$LCSL(P, Q) = \max(LCSL(P[1 \cdots n-1], Q), \\ LCSL(P, Q[1 \cdots m-1]) \\ LCSL(P[1 \cdots n-1], Q[1 \cdots m-1]))$$

证明:

如果 $p_n = q_m$, 那么 p_n和 q_m 一定在 optimal solution matched together

↘

先证明 p_n和 q_m 至少有一个是在 optimal solution 中 不一定被 matched together

∵ if there is an optimal solution doesn't include p_n or q_m.
we can construct a new optimal solution O^* such that p_n and q_m are matched together

∴ $|O| < |O^*|$

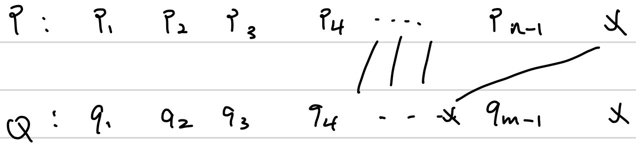
∴ Contradiction !

Therefore, at least one occurrence of p_n / q_m in optimal solution

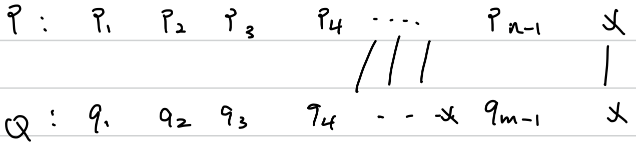
Proof continue:

Suppose p_n and q_m does not match in the optimal solution

∴ optimal solution looks like: (suppose $p_n = q_m = x$)


P: p_1 p_2 p_3 p_4 .... p_{n-1} x
Q: q_1 q_2 q_3 q_4 - - x q_{m-1} x

we can construct another optimal solution O^* looks like


P: p_1 p_2 p_3 p_4 .... p_{n-1} x
Q: q_1 q_2 q_3 q_4 - - x q_{m-1} x

$|O| = |O^*|$

O^* is also an optimal solution

∴ when $p_n = q_m$, p_n and q_m will be in optimal solution and will be matched together

上面的证明证明了第一个case

else part:

P: p_1 p_2 p_3 p_4 ...... p_{n-1} x

Q: q_1 q_2 q_3 q_4 ..... q_{m-1} y

如果 $p_n \neq q_m$:

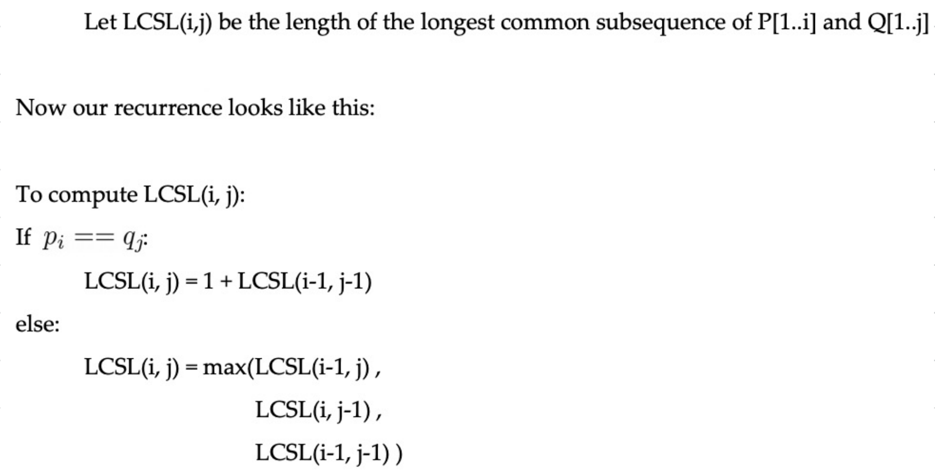
# 要么 p_n 和 q_j $(j < m)$ 相连，所以 q_m 可以放弃

# 要么 p_j $(j < n)$ 和 q_m 相连，所以 p_n 可以放弃

# 要么两个都不处于 solution 中，两个均可放弃

以上三个 case 均可把问题化成 subproblem

稍加修改，变成最终的：



Let LCSL(i,j) be the length of the longest common subsequence of P[1..i] and Q[1..j]

Now our recurrence looks like this:

To compute LCSL(i, j):

If $p_i == q_j$:

LCSL(i, j) = 1 + LCSL(i-1, j-1)

else:

LCSL(i, j) = max(LCSL(i-1, j) ,
LCSL(i, j-1) ,
LCSL(i-1, j-1))

同时，再定义几个 base case：

LCSL (1, 1) = 1 if $p_1 == q_1$
= 0 otherwise

LCSL (1, j) = 1 if LCSL (1, j-1) == 1 or $p_1 == q_j$ for all j > 1
= 0 otherwise

LCSL (i, 1) = 1 if LCSL (i-1, 1) == 1 or $p_i == q_1$ for all i > 1
= 0 otherwise

并以此构建表格：

	M	A	I	L	R	O	O	M
P	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
L	0	1	1	2	2	2	2	2
I	0	1	2	2	2	2	2	2
N	0	1	2	2	2	2	2	2
D	0	1	2	2	2	2	2	2
R	0	1	2	2	3	3	3	3
O	0	1	2	2	3	4	4	4
M	1	1	2	2	3	4	4	5
E	1	1	2	2	3	4	4	5

M

+1

For instance: represent the length of LCS between "PALINDROME" and "MA" only equals to 1

回溯：

1. Store additional information

2. Reconstruct:

Start from optimal num 5:

1. 在 5 时，E ≠ M → 是由别处来的
2. 走向上面，M = M → 同时向左和上走

......

......

Complexity: O(n * m)

问题五： Subset Sum

↳ Given a set S of n integers and a target value K, does S have a subset that sums to K?

↳ NP - Complete

↳之前用 divide and conquer (Pair sum 啥的)

↓

$O(n * 2^{\frac{n}{2}})$

Dynamic Programming Algorithm

首先，作一个假设： Set 里面只有 positive integer

Given a set S of n positive integers and a target value K, does S have a subset that sums to K?

↳

① Sort the positive integers in S in ascending order

$S = \{s_1, s_2, s_3 \cdots s_n\}$

② SubSum(n, K) gives a True if $\{s_1, s_2, \cdots s_n\}$ has a subset that sums to K.

* E.g. $S = \{1, 4, 7, 9, 12\}$, $K = 14$

* SubSum (5, 14) = True

* The subset that sums to 14 is {1, 4, 9}

↳ Sn 要么在 Subset that sums to K，要么不在 Subset that sum to K

$$\begin{cases} \text{SubSum}(n, K) = T & \text{if } \text{SubSum}(n-1, K-s_n) = T \\ \text{SubSum}(n, K) = T & \text{if } \text{SubSum}(n-1, K) = T \end{cases}$$

$$\therefore \text{SubSum}(n, K) = \begin{cases} \text{SubSum}(n-1, K-S_n) \\ \text{SubSum}(n-1, K) \end{cases}$$

Set

↓

K

↳ 换个 Parameter，更 generalize

$\text{SubSum}(i, x) = \text{SubSum}(i-1, x),$
$\text{SubSum}(i-1, x-S_i)$

$\{[a, b, c, D] \boxed{E}\}$

$$\begin{cases} \text{SubSum}(n-1, \underline{K - S_n}) \\ \text{SubSum}(n-1, K) \end{cases}$$

Base Case:

1、 SubSum$(i, 0)$ = T   # K=0 的话，一个空集即可

2、 SubSum$(1, x)$ = T , if $s_1 = x$

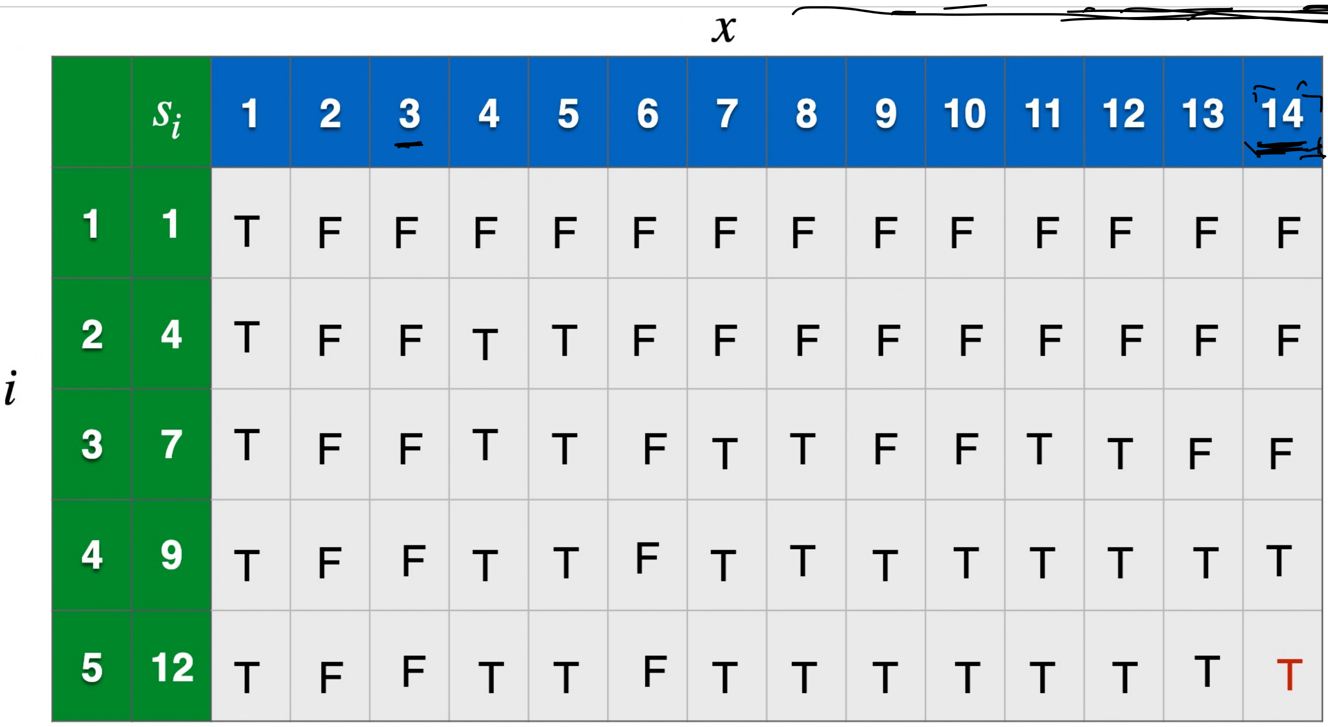
SubSum$(1, x)$ = F , otherwise

3、 SubSum(i, x) = F if $x < 0$

4、 SubSum(i, x) = T, if $s_i = x$

$$\text{SubSum}(i, x) = \text{SubSum}(i-1, x),$$
$$\text{Subsum}(i-1, x - s_i)$$

↳ 可以画一张图，i.e 用一个 data structure 存下来，$S = \{1, 4, 7, 9, 12$. $K = 14$



x

i	s_i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	4	T	F	F	T	T	F	F	F	F	F	F	F	F	F
3	7	T	F	F	T	T	F	T	T	F	F	T	T	F	F
4	9	T	F	F	T	T	F	T	T	T	T	T	T	T	T
5	12	T	F	F	T	T	F	T	T	T	T	T	T	T	T

*. 回溯

↳ 直接 reconstruct / 在 construct 的时候用 additional information

Complexity: 创建一个 $n*K$ 的表格，但，如果 K很大的话，就很凉

↳ if $K = 2^n$, then $O(n \times 2^n)$

Matrix chain multiplication

↳ Given a sequence $\langle A_1, A_2, \cdots A_n \rangle$ of n matrices

↳ wish to compute $A_1 \cdot A_2 \cdot A_3 \cdots \cdot A_n$

↓

怎么归类会带来巨大的影响：

$(A_1(A_2(A_3A_4)))$ $(A_1((A_2A_3)A_4))$ $((A_1A_2)(A_3A_4))$ $((A_1(A_2A_3)A_4)$ $(((A_1A_2)A_3)A_4)$

其次，以下代码是矩阵相乘的代码

```
Matrix-Multiply (A, B) :
    if A.columns ≠ B.rows:
        error "invalid"
    else :
        let C be a new array with size A.rows × B.columns
        for i = 1 to A.rows :
            for j = 1 to B.columns :
                $c_{ij} = 0$  → 一定要先initialize，不然下面的loop很难算
                for k = 1 to A.columns
                    $\boxed{c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}}$  → 计算实际发生
    return C
```

假设：$A_1 : 10 \times 100$, $A_2 : 100 \times 5$, $A_3 : 5 \times 50$

$((A_1A_2)A_3)$

↳ $A_1A_2 \Rightarrow 10 \times 5 \times 100 = 5000$ 次计算

↓ +

$A_1A_2(10 \times 5) \cdot A_3 = 10 \times 5 \times 50 = 2500$次计算

‖

7500次计算

$(A_1(A_2A_3))$

↳ $A_2A_3 \Rightarrow 100 \times 5 \times 50 = 25000$次计算

↓ +

$A_1 \cdot A_2A_3(100 \times 50) = 10 \times 100 \times 50 = 50000$ 次计算

‖

75000

10 times more calculation ⟵

∴ 问题变成了一个优化问题：

Matrix-chain multiplication problem: Given a chain $\langle A_1, A_2 \ldots, A_n \rangle$ of n matrices, where $i = 1, 2, \cdots n$ matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1, A_2 \cdots A_n$ in a way that minimizes the number of <u>calculation</u>

Recurrence Relation: $P(n)$ indicate the number of alternative parenthesizations of a sequence of n matrices by $P(n)$

Base case: $n=1$ → 只有一种办法，所以 $P(1)=1$

$n \geq 2$ → product of two fully parenthesized matrix subproducts

↳ 因为 $P(n)$ 指全部的可能

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} \underbrace{P(k)}_{} P(n-k) & \text{if } n \geq 2 \end{cases}$$

→ $k = 1, 2, \cdots, n-1$ (把在最后分割的可能全加起来)

Applying dynamic programming:

1、characterize the structure for an optimal solution

2、Recursively define the value of an optimal solution

3、Compute the value of an optimal solution, typically in a bottom-up fashion

4、Construct an optimal solution from computed information

↳ 之前讲的4个步骤

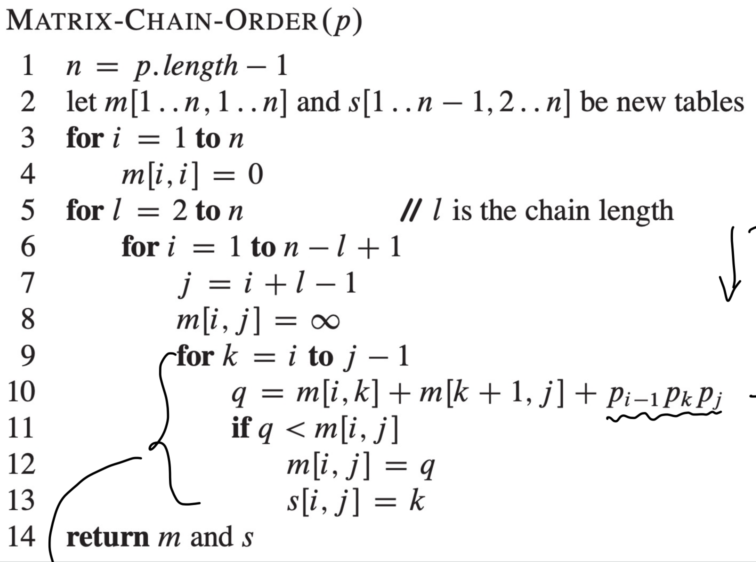
1、就很直观，就找到 $\langle A_1 A_2 \cdots A_n \rangle$ 的 optimal solution

= 找到 $\langle A_1 A_2 \ldots A_k \rangle$ 和 $\langle A_{k+1} A_{k+2} \ldots A_n \rangle$ 各自的 optimal solution 并 combine

2、定义：m[i,j] be the minimum number of scalar multiplications needed to compute the matrix $A_{i \ldots j}$

$$m[i,j] = \begin{cases} 0 & \text{if } i=j \\ \min\limits_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & \text{if } i<j \end{cases}$$

$s[i,j] = k$ → 用来 reconstruct 的



```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i,i] = 0
for l = 2 to n              // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i,j] = ∞
        for k = i to j − 1
            q = m[i,k] + m[k+1,j] + p_{i−1} p_k p_j
            if q < m[i,j]
                m[i,j] = q
                s[i,j] = k
return m and s
```

→ 大意也是遍历全部选择，为了找到 minimum 的 order

→ P这里定义的是要把 m[i,k] 和 m[k+1,j] 合并在一起的数据

→ 把所有k都遍历一遍 找到最小的

4、Reconstruct

其实因为存了 S[i,j]，reconstruct 的过程很简单

CISC/CMPE-365*
Test #3
November 1, 2013

Student Number (Required) ______________________

Name (Optional)________________________________

This is a closed book test. You may not refer to any resources.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, but will not be reconsidered after the test papers have been returned.

The test will be marked out of 50.

Question 1	/30
Question 2	/20
TOTAL	/50

I guess the issue for me is to keep things dynamic.

Robert Downey, Jr.

Question 1 (30 marks)

The President of Elbonia, impressed by your ability to stack concrete blocks, has put you in charge of packing a large container full of national treasures which he plans to take with him for “safe-keeping” on his upcoming trip to Switzerland.

The container can hold at most k kilograms. Each treasure t_i has a value v_i and a mass m_i. Your task is to find the most valuable combination of treasures that will fit in the container.

For example, if k = 10 and the table of values and masses looks like this

	t_1	t_2 ✓	t_3	t_4 ✓	t_5
Value	210 kronks	200 k.	150 k.	75 k.	24 k.
Mass	8	5	6	3	4

then the optimal solution is to take t_2 and t_4

MV(2, ~~7~~ y)

y

Create a dynamic programming solution for this problem. Here is a definition that may be useful:

Let MV(i,x) be the value of the most valuable subset of $\{t_1 \ldots t_i\}$ such that the total mass of the selected treasures is <= x

a) Characterize the solution as a sequence of decisions

b) Show that the problem satisfies the Principle of Optimality

c) Give a complete recurrence relation for the problem

d) Describe the order in which you will compute the solutions to sub-problems

e) Explain how you will extract the details of the optimal solution

(Write your answer on the next page)

1.

$$\begin{tikzcd} & \text{if } m_i \le x \arrow[r] & \text{take } t_1 \\ MV(i, x),\ i=1 \arrow[ur] \arrow[r] & \text{else, output } 0 & \end{tikzcd}$$

$MV(i, 0) = $ float("inf")

$$MV(i, x) = \max(V_i + MV(i-1, x-m_i),$$
$$MV(i-1, x))$$

(d) Final answer will be 2-dimensional array such that row represent the increasing of element and column represent increasing of x value. When building up answer, we fill in one row of all column increasingly.

(e) The optimal solution will be at $MV[i][x]$ ie. the bottom right position. Given this value we first examine if $MV[i-1][x-m_i]$ have the same value as $MV[i][x]$, if yes, t_i is not in the optimal set, else it's in. Or we can just record more information while building up solution

2.

$P(i, j, x) = P(i-1, j, x) + W((i-1, j), (i, j))$ if $j == 0$

$P(i, j, x) = P(i, j-1, x) + W((i, j-1), (i, j))$ if $i == 0$

Question 2 (20 Marks)

The President of Elbonia has been arrested on his way to the airport, and as his assistant you are wanted for questioning. Your escape plan is to walk from Elbonia to the neighbouring country of Dorkis. The paths between the two nations form a rectangular grid with n rows and m columns. You are at the top left corner of the grid and your destination is at the bottom right corner. All of the horizontal path segments run left-to-right, and all of the vertical segments run top-to-bottom. Each path segment has a value attached to it that represents the time required to walk that segment.

So far, this is identical to the problem we examined in class. Here is the difference: some of the path intersections are known to have toll-booths, charging 10 kronks for passage. You only have k 10-kronk coins in your pocket, so any path that includes more than k toll-booths cannot be used. (For example, if k = 3, you can pass through up to three toll-booths, but no more.)

You may assume that there are no toll-booths along the top edge, or along the right-hand side of the grid, so it is possible to reach the goal without passing through any toll-booths at all. Your mission is to find the fastest route that passes through at most k toll-booths.

Here is part of a recurrence relation:

Let P(i,j,x) be the length of the shortest path from the starting point (0,0) to intersection (i,j), passing through at most x toll-booths.

当 x =0时，已经没有可供花费的 kronks

直接不选

P(i,j,0) = infinity if intersection (i,j) is a toll-booth

= min{P(i, j-1, 0) + w(edge from i,j-1 to i,j),

P(i-1, j, 0) + w(edge from i-1,j to i,j)} if intersection (i,j) is not a toll-booth

for x > 0

P(i,j,x) = min{ P(i, j-1, x-1) + w(edge from i,j-1 to i,j) ,

P(i-1, j, x-1) + w(edge from i-1,j to i,j)} if intersection (i,j) is a toll-booth

= min{ P(i, j-1, x) + w(edge from i,j-1 to i,j),

P(i-1, j, x) + w(edge from i-1,j to i,j)} if intersection (i,j) is not a toll-booth

a) Complete this recurrence relation by adding appropriate base cases. For convenience, here is the recurrence again:

P(i,j,0) = infinity if intersection (i,j) is a toll-booth

= min{P(i, j-1, 0) + w((i,j-1),(i,j)),

P(i-1, j, 0) + w((i-1,j),(i,j)) } if intersection (i,j) is not a toll-booth

for x > 0

P(i,j,x) = min{ P(i, j-1, x-1) + w((i,j-1),(i,j)),

P(i-1, j, x-1) + w((i-1,j),(i,j))} if intersection (i,j) is a toll-booth

= min{ P(i, j-1, x) + w((i,j-1),(i,j)),

P(i-1, j, x) + w((i-1,j),(i,j))} if intersection (i,j) is not a toll-booth

Base Case :

$P(i, j, x) = P(i-1, j, x) + w((i-1, j), (i, j))$ if $j == 0$

$P(i, j, x) = P(i, j-1, x) + w((i, j-1), (i, j))$ if $i == 0$

b) Explain the order in which you will compute the solutions to sub-problems.

To compute the solution to this problem, the solution consist of 3 dimension : i, j, k

Sol_list [i] [j] [k] given length of shortest path from (0, 0) to (i, j) with at most k kronks.

we fill out the array by at first k=0 -> row by row

2014

Question 1 (30 marks) → 问题基本一样，多了一个限制条件

↳ 只需多给一个 base case 就 ok

After graduation you find yourself working in a steel mill, ironically named Dynamic Industries. The mill produces steel bars in a variety of lengths and the bars are then cut into shorter lengths for sale. Your job is to determine how to cut the bars so as to maximize the total sale value. Ho hum, we did that in 365 with Prof. Whats-his-name. But wait! This is different! Now you don't have unlimited access to the bar-cutting saw. For each bar, you are told the length of the bar and also the maximum number of cuts you can make.

For example, suppose the sale value for pieces of length 1 through 5 are given by this table:

Length	1	2	3	4	5
Value	2	3	5	7	8

If you are given a bar of length 5 and you are allowed to make 3 cuts, then you can make

0 cuts, for a value of 8, or

1 cut (perhaps into a 1 and a 4, for a value of 9) or

2 cuts (perhaps a 1 and two 2's, for a value of 8) or

3 cuts (perhaps three 1's and a 2, for a value of 9).

Consider the following recurrence relation, which is based on the “leftmost cut” method:

Let MV(n,k) represent the optimal value we can get from a bar of length n, using no more than k cuts.

```
MV(n,0) = Value(n)                          #no cuts allowed

MV(1,k) = Value(1)                          #can't subdivide a piece of length 1

MV(n,k) = max{ Value(n),                    #no cut

               Value(1) + MV(n-1,k-1),      #leftmost cut at 1

               Value(2) + MV(n-2,k-1),      #leftmost cut at 2

               Value(3) + MV(n-3,k-1),      #etc.

               ...

               Value(n-1) + MV(1,k-1)

             }
```

a) (7 marks) Show that the problem satisfies the Principle of Optimality.

8

Suppose that in some optimal solution S, the leftmost cut is at i. Then the remaining set of cuts in S are a solution to the reduced problem of cutting a bar of length n-i, using at most k-1 cuts.

Suppose there is a better solution to this reduced problem. Then we could combine this better solution with the leftmost cut of S to get a solution that is better than S, which contradicts the optimality of S.

Thus the embedded solution to the subproblem of cutting a bar of length n-i using at most k-1 cuts is optimal. Thus the Principle of Optimality is satisfied.

Marking: the key concept is that an optimal solution must contain only optimal solutions to subproblems. If the student shows that they understand this, they should get at least 4/7

b) (7 marks) Design a table to hold information about optimal solutions to subproblems.

Use a 2-dimensional table MVT with "lengths": 1..n as labels on the rows and "number of cuts allowed": 0..k as labels on the columns (or vice versa). Use MVT[i,j] to store MV(i,j) – ie the optimal value of a bar of length i, using at most j cuts

Marking: they should remember that they need a column (or row, if they transpose the table) for "0 cuts" - take off a couple of marks if they forget this. Again, the key concept is creating a table to store results so that nothing needs to be calculated twice. If they show understanding of this, they should get at least 4/7

Some students may choose to store more information in the table, such as the cuts that have been used to achieve the optimal solutions. This is not a problem.

c) (7 marks) Describe the order in which you will compute the solutions to subproblems, and why.

Observe that MV(1,j) = Value(1) for all j. Thus we can fill in the first row of the table immediately. After this, fill in the table row by row, since each MV value depends only on values from previous rows.

An argument can also be made for filling in the values column by column, since each MV value only depends on values from the previous column.

Marking: the key concept is computing solutions to subproblems in a logical order so that when MV(i,j) is to be computed, all the relevant smaller problems have already been solved. Understanding that is worth 4/7, even if they are unable to give an effective order.

Some students may take a recursive, top-down approach. This is ok, although it makes it difficult to describe the exact order in which subproblems will be solved. In this case the answer will probably be that subproblem solutions will be be computed in an "as-needed" order, which is pretty much self-explanatory.

这是讲了个废话：可以从 MVT[n, k] 处，re iterate 所有之前的组合

或 在创建 MVT时 多储存些信息

d) (7 marks) Explain how you will extract the details of the optimal solution.

Once the table is full, the maximum possible value obtainable will be the value in MVT[n,k] (the bottom right hand corner of the table). Starting from this point, we can re-evaluate all the possible predecessors of this table element, and determine which one led to the final optimal value. This gives us the position of the final cut. Then we repeat the process to work back from there, until we have determined each cut in the optimal solution.

Alternatively, every time we compute MVT[i,j], we could record the option that gave us this value. Then we can trace back from MVT[n,k] without having to re-evaluate the possible predecessors.

Marking: Students' answers will depend on what information they choose to store in the table, but they should give a good explanation of whatever is appropriate for the table they described. If they are all at sea but they give enough of an answer to show they understand the idea of tracing back from the final table entry, they should get at least 4/7

Question 2 (8 Marks)

Let S be a set of n positive integers, with n >= 1

Let k be a positive integer such that k <= 1000

What is the computational complexity of solving the Subset Sum problem on S and k, using the algorithm that we developed in class?

*The algorithm we used creates a table that is n*k in size, and fills each element of the table in constant time. Since we know k <= 1000, the table has <= 1000*n elements. Filling them in takes O(n) time.*

Marking: if they state that it is polynomial but not O(n), they should get 5/8. If they state it is not polynomial because Subset Sum is NP-Complete, they should get 2/8.

Subset Sum (n, k) 创建了 一个 n*k 的 Array
这里给 k定了 上限 k<= 1000
$\therefore O(1000\,n) = O(n)$

Question 3 (14 Marks)

You are visiting Aggravatia, where the currency is based on coins of value {1, 4, 7, 9}. Nobody in the country has been able to solve the change-making problem: given a target value k, find the smallest set of coins that sums to k. The Minister of Finance offers you the job of creating a Dynamic Programming solution.

Define CM(k) = the minimum number of coins needed to sum to k, where k >= 0. For example, CM(11) = 2, since 11 = 4+7

Here is part of a recurrence relation for CM(k):

$CM(k) = 1 + \mathbf{min}\{CM(k-1),CM(k-4),CM(k-7),CM(k-9)\}$ if $k >= 9$

a) (8 marks) Complete this recurrence relation by adding appropriate formulae for all remaining cases. 把全部的 Base case 加进去

$CM(k) = 1 + min\{CM(k-1),CM(k-4),CM(k-7)\}$ if $7 <= k < 9$

$CM(k) = 1 + min(CM(k-1),CM(k-4)\}$ if $4 <= k < 7$

$CM(k) = 1 + CM(k-1)$ if $1 <= k < 4$

$CM(0) = 0$

Marking: 2 marks for each line. If they do it a completely different way, give marks as seem appropriate, depending on correctness. If they show they know the purpose of the recurrence relation to describe a relation between the optimal solution for k and the optimal solutions of smaller problems, they should get at least 4/8

b) (6 marks) Determine the computational complexity of computing CM(k) for k >= 0

We can use the recurrence to compute CM(i) for i = 1..k. Each computation is based on at most 4 previously computed CM values, so the algorithm runs in O(k) time.

Marking: If they say polynomial but not O(k), they should get 3/6
Some students may say "linear" or O(n) – that's fine.

CISC/CMPE-365*
Test #3
November 5, 2015

Student Number (Required) 20060593

Name (Optional)

This is a closed book test. You may not refer to any resources.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, but will not be reconsidered after the test papers have been returned.

The test will be marked out of 50.

Question 1	/25
Question 2	/25
TOTAL	/50

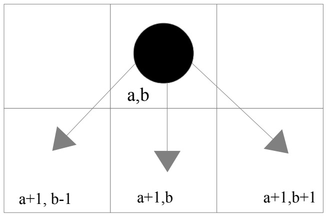
Remember, remember, the fifth of November

Question 1 (25 Marks)

Suppose that you are given an n * n checker board and a single checker.

You must move the checker from the top row (row 1) of the board to the bottom row (row n). At each step, you may move the checker to one of the following squares:

- the square one row down and one column to the left, if there is one – i.e. diagonally down to the left
- the square one row down in the same column – i.e. immediately below
- the square one row down and one column to the right, if there is one – i.e. diagonally down to the right



看清楚，花费发在在 square

We will use the notation **[a,b]** to represent the square in row a and column b, so from [a,b] you can move to [a+1,b-1] or [a+1,b] or [a+1,b+1], as shown in the diagram.

Each square contains a quantity of money – the value of square [a,b] is given by Value(a,b).

You are allowed to start on any square in the top row and finish on any square in the bottom row. Your task is to create a Dynamic Programming algorithm to find the path from the top to the bottom with the **maximum** total value.

Example: consider this 3*3 board. The values are shown in each square, and the optimal path is highlighted in grey (the other squares have all been left white for clarity).

6	2	3
8	3	6
4	5	9

不管你之后的选择是什么，Value(a,b) 是不得不算上的

Define MV(a,b) to be the value of the optimal path from square [a,b] to the bottom row.

$$MV(a,b) = \underbrace{Value(a,b)}_{} + Min(MV(a+1, b-1), MV(a+1, b), MV(a+1, b+1))$$

顶到边界了(左边)

$$\text{Special case: } MV(a,b) = Value(a,b) + Min(MV(a+1, b), MV(a+1, b+1))$$

$$MV(a,b) = value(a,b) + Min(MV(a+1, b), MV(a+1, b-1))$$

(a) **[10 marks]** Here is part of a possible recurrence relation for MV(a,b)

$$
\begin{aligned}
MV(a,b) = Value(a,b) + \max\{\ & MV(a+1, b-1),\\
& MV(a+1,b),\\
& MV(a+1,b+1)\\
\}&
\end{aligned}
$$

Complete the definition of this recurrence relation, or substitute your own complete recurrence relation if you prefer. Think about base cases. Think about special cases when you are at the left or right side of the board.

Solution:

Base cases: MV(n,b) = Value(n,b) for all b

都是顶到边界了

↑

Special cases:

MV(a,1) = Value (a,1) + max{MV(a+1,1), MV(a+1,2)} *# there is no column 0*

MV(a,n) = Value(a,n) + max{MV(a+1,n-1), MV(a+1,n)} *# there is no column n+1*

Marking:

Base cases: 5 marks

Special cases: 5 marks

If the student clearly understood what was required but could not properly solve the base cases and special cases, they should get at least 6/10

Two dimensional array

(b) **[5 marks]** What data structure will you use to store the MV() values?

↳ 可以观察题目提供的条件：用多少个参数就代表有多少个维度

Solution:

The natural choice is a 2-dimensional array with the same dimensions as the board. Let MVT be this array. Then MV(a,b) will be stored in MVT[a][b]

Marking:

I'm not sure what alternative answers might be given ... but the important idea is that we need to be able to access each MV() value in constant time. If the student describes a structure with that in mind, they should get at least 3/5

row by row

(c) **[5 marks]** In what order will you compute the MV() values?

Solution:

Using the base cases, we can compute MV(n,b) for all b

Then we can compute MV(n-1,b) for all b, then MV(n-2,b) for all b, etc

The last values computed would be MV(1,b) for all b

Marking:

The key concept is that the MV() values must be computed in an order that makes sure all required information is available when it is needed.

☆☆

Question 2 (25 Marks)

Consider the 0/1 Knapsack Problem: Given a set of n objects $\mathbf{S = \{s_1, ..., s_n\}}$, each with mass $\mathbf{m_i}$ and value $\mathbf{v_i}$, and a container with capacity **k**, we want to find the maximum-value subset of the objects that will fit in the container.

A dynamic programming solution for this problem may be created using a recurrence relation like this:

Let KS(i,x) = the maximum value we can obtain from $\{s_1, ..., s_i\}$ with a container of capacity x

```
if $m_i > x$        KS(i,x) = KS(i-1,x)                  # if $s_i$ is too big, we can't take it

if $m_i <= x$       KS(i,x) = max{ vi + KS(i-1, x – $m_i$),    # we either take $s_i$
                               KS(i-1, x)                 # or we don't
                             }
```

with base case
KS(1,x) = v_1 if x >= m_1
= 0 if x < m_1

Now suppose a further constraint is added: **we can't choose more than r objects**, where r is any integer.

(a) **[15 marks]** Revise the recurrence relation to adapt to this modification.

Solution: We can add a third parameter to the recurrence relation to indicate the number of objects we are permitted to take. Each time we take an object, this number decreases.

Use KSL(i,x,t) to represent the maximum value we can obtain from $\{s_1, ..., s_i\}$ with a container of capacity x and an object limit of t

Question 2 (25 Marks)

Consider the 0/1 Knapsack Problem: Given a set of n objects **S = {s_1, ..., s_n}** , each with mass **m_i** and value **v_i**, and a container with capacity **k**, we want to find the maximum-value subset of the objects that will fit in the container.

A dynamic programming solution for this problem may be created using a recurrence relation like this:

Let KS(i,x) = the maximum value we can obtain from {s_1, ..., s_i} with a container of capacity x

```
if $m_i > x$        KS(i,x) = KS(i-1,x)              # if $s_i$ is too big, we can't take it

if $m_i <= x$       KS(i,x) = max{ vi + KS(i-1, x – $m_i$),      # we either take $s_i$
                                   KS(i-1, x)                    # or we don't
                                 }
```

with base case
KS(1,x) = v_1 if x >= m_1
= 0 if x < m_1

Now suppose a further constraint is added: **we can't choose more than r objects**, where r is any integer.

KS(i, x, r) : the maximum value we can obtain from $\{S_1, ..., S_i\}$ with a container of capacity x

KS(i, x, 0) = 0 # Can't take anything

if $m_i > x$, KS(i, x, r) = KS(i-1, x, r)

if $m_i < x$, KS(i, x, r) = max(KS(i-1, x - m_i, r-1) + V_i ,
KS(i-1, x, r))

KS(1, x, r) = V_1 if $m_1 < x$
0 if $m_1 > x$

Then the recurrence becomes

```
if $m_i > x$          $KSL(i,x,t) = KSL(i-1,x,t)$                # if $s_i$ is too big, we can't take it

if $m_i <= x$ and $t > 0$
                      $KSL(i,x,t) = max\{ vi + KSL(i-1, x - m_i,t-1),$       # we either take $s_i$
                                  $KSL(i-1, x,t)$                           # or we don't
                               }
```

with base cases

$KSL(i,x,0)$ $= 0$ *for all* i *and* x

$KSL(1,x,t)$ $= v_1$ *if* $x >= m_1$ *and* $t > 0$
$= 0$ *if* $x < m_1$

Marking:

The essential concept is the inclusion of a third parameter that reduces as items are selected. A student who does this should receive at least 8/15. If a student gets the recursive part of the recurrence correct but does not get the base cases, or vice versa, they should get at least 10/15. A student who gets both parts almost correct should get 13/15 or 14/15

(b) **[5 marks]** What is the complexity of computing each KS() value in your revised recurrence relation (assuming the relevant subproblems have already been computed)?

$O(n^3)$

一定要看清楚题目

Solution:

Each value is computed in constant time since there are fixed number of relevant subproblems.

???

Marking:

It is not technically incorrect to answer "O(n)" or even "polynomial" but these are weak answers, only true because "constant time" is included in O(n^t) for all t >= 0. Students who answer in either of these ways should get 2.5/5

(c) **[5 marks]** What is the complexity of computing the entire collection of KS() values?

Solution: * 所以在写 complexity 的时候，最好先写上 本来的变量名

*The total number of values we need to compute is n*k*r and each one takes constant time, so the complexity is O(n*k*r)*

$O(n^3)$

For ***2 bonus marks****, we can observe that values of r that exceed n can simply be reduced to n, since we cannot possibly take more than n items. This gives O(n^2 * k)*

Marking:

If the student understands that we need to multiply the number of values to be computed by the time to compute each one, they should get at least 3/5

Students who respond that the complexity is polynomial should get 1/5, since they are claiming that this is a polynomial-time algorithm for an NP-Complete (technically, NP-Hard) problem.

CISC-365*
Test #3
February 12, 2019

Student Number (Required) ______________________

Name (Optional)________________________________

This is a closed book test. You may not refer to any resources.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, **but will not be re-marked under any circumstances.**

The test will be marked out of 50.

Question 1	/28
Question 2	/20
Question 3	/2
TOTAL	/50

Question 1 (28 marks)

Congratulations! Your international prestige as a problem-solver has earned you a new job – you now operate a guided-tour business in Balatronia.

Tourists sign up for 1-week (Short) or 2-week (Long) guided tours of the local mud pits during the summer season. There is a Short tour and a Long tour starting each week except the last week of the summer - in which there is only a Short tour. Each tour is worth a different amount of tip money, based on the wealth of the tourists. Your goal is to decide which tours to guide personally, without choosing any overlapping tours.

For example, suppose the summer season is 5 weeks long. The tours starting in each week might look like this. Tours are numbered according to the week in which they start.

	Week 1	Week 2	Week 3	Week 4	Week 5
1-week tours	$Short_1$ Value = 10	$Short_2$ Value = 7	$Short_3$ Value = 12	$Short_4$ Value = 4	$Short_5$ Value = 9
2-week tours	$Long_1$ Value = 20		$Long_3$ Value = 18		
		$Long_2$ Value = 22		$Long_4$ Value = 16	

One solution is to choose $Short_1, Long_2, Short_4, Short_5$ with a total value of 45

A better solution is to choose $Long_1, Short_3, Long_4$ with a total value of 48

In Week 1, you can guide either the 1-week tour ($Short_1$) or the 2-week tour ($Long_1$). In Week 2, you are either halfway through tour $Long_1$ or you can start guiding either of the tours that start in Week 2 (if you chose $Short_1$ in Week 1).

This question asks you to construct a Dynamic Programming solution to maximize your personal profit. Your solution must work on all instances, not just the example shown here.

$max\_week(1) = Value(Short_1)$ #只有一周的话，只能选择那一周的 Short

$max\_week(2) = max(value(short_1) + value(short_2),$ #只有两周的话，只能从这些选择中选

$value(long_1))$

$max\_week(n) = max(Value(Short\ n) + max\_week(n-1),$

$Value(long_{n-1}) + max\_week(n-2))$

c. Intend to use a table, which row represent the week number and column represent the value. During computation, the table will be filled out start from week1

d. For max.week (n), the optimal value is shown on max_week [n], check $long_{n-1}$ and max_week [n-2] as well as $short_n$ and max-week [n-1], choose the max of them. Then keep iterating through the detail of optimal solution will be revealed.

e. Complexity : O(n) → 直接看 Array 是怎么被建立的就好

(a) (5 marks) Explain how this problem satisfies the Principle of Optimality . Your explanation must be clear but a rigourous proof is not required.

(Hint: Suppose the optimal solution contains a particular tour X. What can you say about the chosen tours that precede X, and the chosen tours that follow X?)

This material was not covered in F2019 … but I have included the solution here in case you are interested!

Solution:

Based on the hint: if tour X is in the optimal solution, then the chosen tours that precede X must be a solution to the subproblem of choosing tours within the weeks before X begins. Similarly, the chosen tours that follow X must be a solution to the subproblem of choosing tours within the weeks after X ends. These must be optimal solutions to these subproblems because if they weren't we could replace them by something better, which would improve the overall optimal solution – which is not possible.

It is also possible to focus on the last tour in the optimal solution – it will be either $Short_n$ ***or*** $Long_{n-1}$***. Whichever it is, the other tours in the optimal solution must be an optimal solution within the weeks preceding the final tour in the optimal solution.***

We could also focus on the first tour in the optimal solution – it will be either $Short_1$ ***or*** $Long_1$ ***. Whichever it is, the other tours in the optimal solution must be an optimal solution within the weeks following the first tour in the optimal solution.***

Marking:

A solution similar to any of the above	5
A solution that shows understanding of the P. of O. without applying it successfully to this problem	3
For trying	1

P(K) =

(b) (8 marks) Give a recurrence relation for this problem.

Hint: Suppose the season is n weeks long. At the end of Week n, you will either be finishing $Short_n$ (and getting its value) or finishing $Long_{n-1}$ (and getting its value). Associate each of these possibilities with the appropriate subproblem. You may want to use "$P(k)$" to represent the maximum profit you can get in the first k weeks of the season.

Solution:

Defining P(k) as above and Val(T) to be the value of tour T, we can use

Recursive part:
for k $\geq$ ***2***
P(k) = max (Val($Short_k$***) + P(k-1),*** ***# finish with a short tour***
Val($Long_{k-1}$***) + P(k-2)*** ***# finish with a long tour***
)

Base cases:
P(0) = 0
P(1) = Val($Short_1$***)***

Marking:

for a correct recursive part	5
for a partially correct recursive part	3
for trying	1
for a correct base	3
for a partially correct base	2
for trying	1

Students might omit the P(0) = 0 base case. That's ok but their recursive part has to be written in such a way that it never tries to recurse to P(0) … so if it refers to P(k-2), it must ensure that k > 2.

(c) (5 marks) Explain and justify the order in which you will compute solutions to subproblems. If you plan to use a table to store solutions to subproblems, this is the place to describe it.

Solution:

Since the recurrence relation only has one parameter, we can use a 1-dimensional array A to hold the solutions to the subproblems : A[i] will be used to store the value of P(i). The array should be indexed from 0 to n. The array is initialized with A[0] = 0, A[1] = P(1).

After that, the elements of the array are filled in ascending order. Each element's value depends on the two values immediately to its left. This order is chosen because it traverses the array in a natural manner and each element's value is computed as soon as the information needed is available.

It is also acceptable to manage the filling of the table using recursion (or a stack!) to keep track of the subproblems encountered. Each subproblem is encountered multiple times but solved only once. Due to the nature of this particular problem, the table will still be filled in from left to right!

Marking:

For any rational plan for the order of solving the subproblems	5
For an explanation with minor/significant/major errors	4/3/2
For trying	1

(d) (5 marks) Explain how you will determine the details of the optimal solution.

Solution:

When we know the optimal final value, we can look at the two values immediately to its left in A to determine which of those options led to the optimal answer. This tells us whether we ended with a Short or Long tour. From whichever element of A led to the final answer, we repeat this process to determine the tour we choose before the last one ... and so on back to the start of the summer.

Alternatively, the table could have been defined to also contain information regarding the elements of the optimal solution. In that case, the extraction of this information would be based on how it was stored.

Marking:

For a reasonably clear explanation of how to get the information	5
For an explanation with minor/significant/major errors	4/3/2
For trying	1

(e) (5 marks) What is the complexity of your algorithm? (Use n to represent the number of weeks in the summer season)

Solution:

Each element of A is computed in constant time, so filling A takes O(n) time.

Each step of the "trace back" is determined in constant time and there are at most n steps, so finding the details of the optimal solution takes O(n) time.

Thus the entire algorithm takes O(n) time.

Marking:

For a correct analysis of their version of the algorithm	5
For an explanation with minor/significant/major errors	4/3/2
For trying	1

QUESTION 2 (20 Marks)

You and your worst enemy are playing a game. Between you are three piles of coins, containing n_1, n_2 and n_3 coins respectively. You take turns removing coins according to this rule: on your turn you must remove a positive number of coins from any one of the piles (ie you must take at least 1 coin). **You win the game if you take the very last coin.**

Each possible game situation is described by the sizes of the piles such as (4,7,2) or (2019,3,12)

If a single move can get from (a, b, c) to (d, e, f) we call (d, e, f) a ***child*** of (a, b, c). For example, we can get from $(8, 7, 5)$ to $(8, 4, 5)$ by removing 3 coins from the centre pile so $(8, 4, 5)$ is a child of $(8, 7, 5)$

We can label a game situation "W" if the player who takes the next turn can be sure of winning, and "L" if they can't. For example $(0, 0, 5)$ is a "W" situation – the player can take the whole third pile, but $(1, 1, 0)$ is an "L" - the player must take 1 coin, then the other player takes the last coin and wins.

In general, a situation is "W" if ***any*** of its children is labelled "L", and a situation is "L" if ***all*** of its children are labelled "W"

Create a recurrence relation to determine if situation (n_1, n_2, n_3) is a "W" or "L"

(Write your answer on the next page)

win-game (n_1, n_2, n_3) :

$$\left.\begin{array}{l} \text{win-game } (x, 0, 0) \\ \text{win-game } (0, x, 0) \\ \text{win-game } (0, 0, x) \end{array}\right\} \rightarrow \text{win}$$

$$\text{win-game } (n_1, n_2, n_3) = \begin{array}{ll} \text{win if } (& \text{win-game } (n_1 - x, n_2, n_3) \text{ lose} \\ & \text{win-game } (n_1, n_2 - x, n_3) \text{ lose} \\ & \text{win-game } (n_1, n_2, n_3 - x) \text{ lose }) \\ \\ \text{lose} & \text{else} \end{array}$$

(a) (10 marks) Recursive part:

Solution:

I will use G(a,b,c) to represent the label of the game when the three piles have sizes a, b, and c.

G(a,b,c) = "W" if G(a,b,x) = "L" for any x in the range [0..c-1] or
G(a,x,c) = "L" for any x in the range [0..b-1] or
G(x,b,c) = "L" for any x in the range [0..a-1]

= "L" if G(a,b,x) = "W" for all x in the range [0..c-1] and
G(a,x,c) = "W" for all x in the range [0..b-1] and
G(x,b,c) = "W" for all x in the range [0..a-1]

The two cases given cover all of the possibilities, so it is not actually necessary to specify both. For example

G(a,b,c) = "W" if G(a,b,x) = "L" for any x in the range [0..c-1] or
G(a,x,c) = "L" for any x in the range [0..b-1] or
G(x,b,c) = "L" for any x in the range [0..a-1]

= "L" otherwise

is perfectly acceptable

Marking:	**- for a correct solution**	**10**
	- for a solution that is mostly correct with a minor error such as making the ranges start at 1 instead of 0	**8**
	- for a solution with a major error such as leaving out one of the sets of subproblems	**6**
	- for a solution that looks like a recurrence relation for the game but is seriously wrong	**4**
	- for a solution that shows limited understanding of what is required	**2**
	- for trying	**1**

(b) (10 marks) Base case(s):

Solution:

The following is sufficient:

$$G(0,0,0) = \text{“L”}$$

but students may include others such as

$$G(0,1,1) = G(1,0,1) = G(1,1,0) = \text{“L”}$$

Students may use other sets of base cases such as

$$G(0,0,x) = G(0,x,0) = G(x,0,0) = \text{“W”} \quad \text{for all } x > 0$$
$$G(0,x,x) = G(x,0,x) = G(x,x,0) = \text{“L”} \quad \text{for all } x \geq 0$$

Another possible answer is (see note below)

$$G(x,y,y) = G(y,x,y) = G(y,y,x) = \text{“W”} \quad \text{for all } x > 0 \text{ and } y \geq 0$$

The important thing is to have a set of base cases such that

- **every possible sequence of moves (eventually) reaches one of the base cases**
- **situations with one or more empty pile are covered**

Note: Students might not include a base case for (0,0,0) since that actually signifies the end of the game. That's ok as long as they "cover" all the states that lead to (0,0,0) so the recursion can't end up at (0,0,0) and not have a resolution. The final answer shown above is an example of such a set of base cases.

Marking:

pretty much the same as Part (a). As noted above, it is important that every sequence of moves in the game ends up in a base case.

QUESTION 3 (2 Marks)

True or False:

It was just a semi-frivolous T/F question. The correct answer was “FALSE”

CMPE/CISC-365*
Quiz #3
November 8, 2019

Student Number (Required) ______________________

Name (Optional)________________________________

This is a closed book test. You may refer to one 8.5 x 11 data sheet.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, **but will not be re-marked under any circumstances.**

The test will be marked out of 50.

Question 1	/24
Question 2	/24
Question 3	/2
TOTAL	/50

"I guess the issue for me is to keep things dynamic."

— Robert Downey Jr.

QUESTION 1 (24 Marks)

You have been chosen to plan a canoe trip down the NottaLottaWatta River for the Queen's University Environmental Exploration Nature Society (acronym: QUEENS) . Canoes are available for rent at trading posts along the river. You will start the trip by renting a canoe at Post 1 (where the river begins) and end the trip in Post n (the end of the river). BUT ... you don't have to keep the same canoe the whole way. You can stop at any post, drop off the canoe you have and rent another one. You can only travel downstream. For all pairs (a, b) with $a < b$, the cost of renting a canoe at Post a and dropping it off at Post b is given by a predetermined matrix Cost(a, b).

For example if there are five posts in total, the costs might be

Cost(a,b) matrix		Post b				
		P_1	P_2	P_3	P_4	P_5
Post a	P_1	x	10	35	50	65
	P_2	x	x	30	35	45
	P_3	x	x	x	15	25
	P_4	x	x	x	x	20
	P_5	x	x	x	x	x

In the example shown, you could rent a canoe from P_1 to P_2, then rent another from P_2 to P_3, then another from P_3 to P_4, then another from P_4 to P_5. This would cost $10 + 30 + 15 + 20 = 75$. Another solution would be to rent a canoe from P_1 to P_3 (cost 35) and another canoe from P_3 to P_5 (cost 25) with a total cost of 60.

Your job is to plan the sequence of canoe rentals to **minimize the total cost.**

We can think of the problem like this: We have to return our last canoe at P_n. We could have rented that canoe at any of $P_1, P_2, \ldots P_{n-1}$. Where-ever we rented our last canoe, we have to solve the rest of the trip optimally from P_1 to that point.

(1) How many different possible solutions are there? Remember there are n posts, where n can be any integer ≥ 2. Explain

↳ Have to rent at P_1,

Can also rent at any subset of $\{P_2, \dots P_{n-1}\}$

number of possible solution

$= \underbrace{2^{n-2}}$

↓ 就是说一个拥有size 为 n 的set，它所拥有的possible subset 数量 $= 2^n$

? ↓

证明1：

从某个集合选子集，每个元素，要么放在子集中

要么不放在子集中 所以每个元素有两种可能

$\therefore 2^n$

证明2：

$\binom{n}{0} + \binom{n}{1} + \dots \binom{n}{n} = 2^n$

(2) Let $MC(i)$ = min cost from P_1 to P_i

$MC(2) = cost(1, 2)$

$MC(i) = \min(Cost(1, i),$

$cost(k, i) + MC(k)$ (for $1 < k < i$))

(c) Complexity : $O(n^2)$

(a) **[6 marks]** How many different possible solutions are there? Remember there are n Posts, where n can be any integer ≥ 2. Explain your answer.

Solution: We have to rent a canoe at P_1***, and we can also rent canoes at any subset of*** $\{P_2, \ldots, P_{n-1}\}$***. Thus the number of possible solutions is the number of subsets of*** $\{P_2, \ldots, P_{n-1}\}$ ***… which is*** 2^{n-2}

Marking:

Correct answer with explanation	**6 marks**
Correct answer without explanation	**4 marks**
"Close" incorrect answer (such as 2^{n-1}) with explanation (such as "any subset")	**3 marks**
"Close" incorrect answer without explanation	**2 marks**
"Wayout" answer (such as n) with or without explanation	**1 mark**

(b) **[12 marks]** Let $MC(i)$ = the minimum cost of getting from P_1 to P_i

(so $MC(n)$ is our over-all solution)

Give a complete statement of a recurrence relation for $MC(i)$.

As a starting point, here is a base case: $MC(2) = Cost(1,2)$

Solution:

for all i > 2:

```
MC(i) = min  ( Cost(1,i),
               MC(i-1) + Cost(i-1,i),
               MC(i-2) + Cost(i-2,i),
               MC(i-3) + Cost(i-3,i),
                ...
               MC(2) + Cost(2,i)
             )
```

Marking:

The hint should suggest that the cost of getting to P_i = the cost of the final canoe that gets us there, plus the minimum cost of getting to the post where we rent that canoe.

The key concept is that the value of MC(i) depends on *all* the previous values.

A student whose answer captures these ideas should get at least 8/12 even if they are unable to correctly express the recurrence relation. Giving 10/12 or 11/12 is appropriate if the answer is close to being correct.

A student whose answer shows that they understand the concept and purpose of a recurrence relation, but not how to create one for this problem, should get at least 6/12

A student whose answer shows only a weak understanding of recurrence relations should get about 3/12

(c) **[6 marks]** Determine the computational complexity of using a Dynamic Programming approach to solve this problem. Explain your answer.

***Solution:** Using the recurrence relation given, the value of MC(i) is computed by taking the min of i-1 values, each of which is computed in constant time. The sum of all computations for MC(n) is thus proportional to the sum 1 + 2 + … + n-1, which is in* $O(n^2)$

Marking:

Same rubric as part (a)

QUESTION 2 (24 Marks)

You have landed a job in a steel mill. The mill produces steel bars of random lengths (all lengths are integers). Strangely, customers seem to prefer steel bars of regular lengths. Your job is cut the raw steel bars into shorter lengths in the most profitable way.

More precisely, you need to cut a bar that is n metres long into shorter pieces, each piece being ≤ 5 metres long. Each short piece has a profit value to the company as shown in this table:

Length	1	2	3	4	5
Profit	2	3	6	9	11

So if n = 6, you could cut the bar into a piece of length 5 and a piece of length 1, with a total profit of 13 … or you could cut the bar into a piece of length 4 and a piece of length 2, with a total profit of 12. There are many other possibilities, including cutting the bar into six pieces of length 1, or two pieces of length 2 and two pieces of length 1, etc.

But if n = 7, cutting a piece of length 5 and a piece of length 2 gives a total profit of 14, while a piece of length 4 and a piece of length 3 gives a total profit of 15. You could also cut the bar into two pieces of size 2 and one piece of size 3, etc. etc.

Design a Dynamic Programming algorithm to find the **maximum profit** obtainable from a bar of length n, where n can be any positive integer.

Hint: remember the dynamic programming algorithm for change-making.

Max Profit (n) = MAX(P_5 + MaxProfit (N-5),
P_4 + MaxProfit (N-4),
P_3 + MaxProfit (N-3),
P_2 + MaxProfit (N-2),
P_1 + MaxProfit (N-1))

要定义那么多 base case 吗 ?

Base: Max Profit (1) = P_1

. . . .

Reconstruct :

length	1	2	3	4	5	6	7	8	9	10
profit	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}

(a) Design a recurrence relation for MaxProfit(n), including base case(s) and a recursive part **[8 marks]**

***Solution:* for each of the lengths between 1 and 5 except 2, the profit cannot be improved by cutting. For a length of 2, we get a better profit (4) by cutting it into two pieces of size 1. So the base cases are:**

MaxProfit(n) = Profit(n) for n = 1, 3, 4, 5
MaxProfit(2) = 4

For n $\geq$ 6, the recurrence relation is:

$$\mathit{MaxProfit}(n) = \max\big(\; 2 + \mathit{MaxProfit}(n-1),\; 3 + \mathit{MaxProfit}(n-2),\; 6 + \mathit{MaxProfit}(n-3),\; 9 + \mathit{MaxProfit}(n-4),\; 11 + \mathit{MaxProfit}(n-5) \;\big)$$

(Note that we can actually leave out the 3+MaxProfit(n-2) option since it will never be optimal … but it's ok to leave it in.)

Marking:

Base Cases:	**3 marks**
Recursive Part:	**5 marks**

As with the recurrence relation part of the previous question, please give part marks if the student understands what is to be done but has some errors in their solution.

(b) Specify how you will store information **[5 marks]**

Solution: ***Since the recurrence relation has only one parameter, we can store information in a 1-dimensional array.***

Marking:

Students might suggest storing the results in a hash-table – it really offers no advantage since we need to solve all the subproblems up to n anyway. I would give 4 marks for this – it's overkill.

Students might also suggest using a 2-dimensional array (I'm not sure how!) - I would give 3 marks for this.

If a student's answer shows that they really didn't understand the concept of storing the results of subproblems in an easily-accessible way, they should get 1 mark for trying.

(c) Specify how you will order your computations **[5 marks]**

***Solution:** MaxProfit(n) depends only on values of MaxProfit(x) where x < n. We can perform the computations from MaxProfit(1) up to MaxProfit(n) – this ensures that all information needed for each MaxProfit value is available when it is needed.*

Marking:

Students may also suggest working from the top down (recursively) and storing each value the first time the subproblem is encountered, then looking the values up on subsequent requests. This is ok – it has the same complexity (just a bit more overhead).

If a student's answer shows that they understand the question but they cannot relate it to this problem, they should get about 2 or 3 out of 5.

(d) Explain how you will reconstruct the set of cuts from the computed MaxProfit(n) information **[6 marks]**

Solution: Once we know the value of MaxProfit(n), we can look at its five possible predecessors (the values for n-1, n-2, n-3, n-4 and n-5) and determine which cut length resulted in the maximum value. This tells us what the final cut was. We work back in this manner to find all the cuts.

Marking:

Students might also suggest "carrying" the optimal set of cuts along in the table, so the solution would be immediately available, or carrying some "most recent cut" information along in which case the solution details can be reconstructed without doing any comparisons. These are both completely acceptable.

An answer which is fundamentally correct but contains some errors should get at least 4/6

If the student's answer shows they understand what is being asked but they can't express a solution for this problem, they should get 2 or 3 out of 6.

QUESTION 3 (2 Marks)

True or **<u>False</u>**:

The 2018 Award for Excellence in Dynamic Programming was won by Netflix.

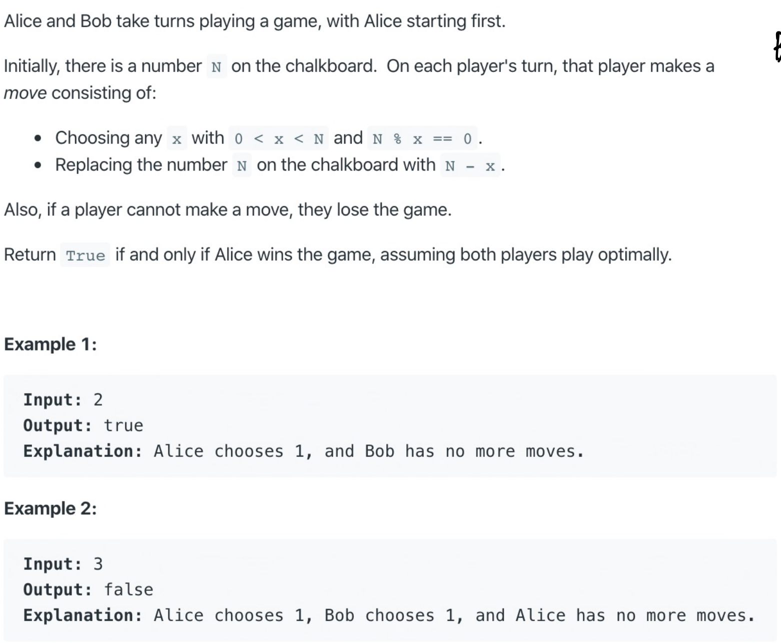
TRUE FALSE

Solution: ***False***

Marking: **2 marks for everyone**

Leetcode

Divisor Game



Alice and Bob take turns playing a game, with Alice starting first.

Initially, there is a number `N` on the chalkboard. On each player's turn, that player makes a *move* consisting of:

- Choosing any `x` with `0 < x < N` and `N % x == 0`.
- Replacing the number `N` on the chalkboard with `N - x`.

Also, if a player cannot make a move, they lose the game.

Return `True` if and only if Alice wins the game, assuming both players play optimally.

Example 1:

```
Input: 2
Output: true
Explanation: Alice chooses 1, and Bob has no more moves.
```

Example 2:

```
Input: 3
Output: false
Explanation: Alice chooses 1, Bob chooses 1, and Alice has no more moves.
```

1. Recurrence Relation:

Base Case: div_Game (N) = False, N = 0, 1
neither of them have any choice

div_Game (N) = (not div_Game (N - i) ||
N > 1 not div_Game (N - i) ......)

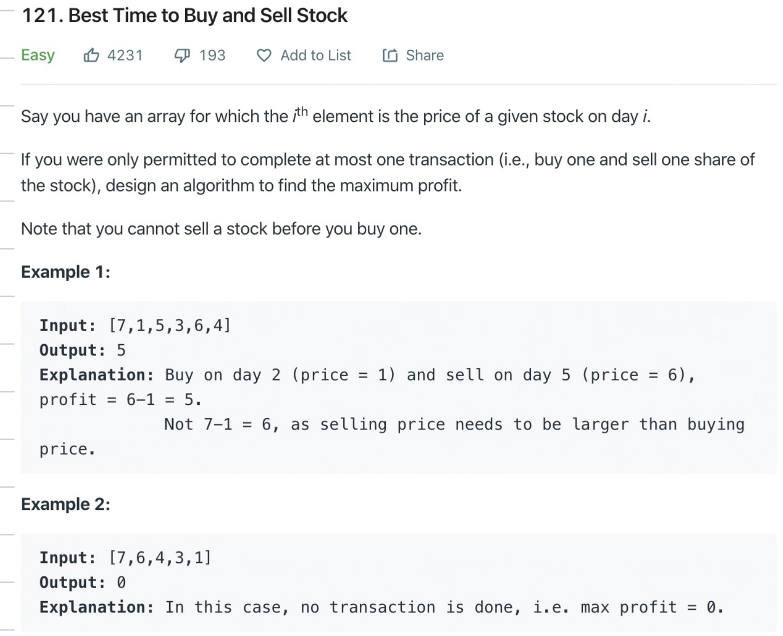
↳ For i < N and N % i == 0

↳ 也就是，对于 N (N > 1)，遍历所有可能的选择，观察 div_Game (N - i) 的值，这个值是对手的，如果找到一个令对手为 False div_Game (N - i) == False 的值，就够了

Implement:

```
def div_Game (N):
    Sol_list = [x for x in range (N+1)]
    Sol_list [0] = False  // Base cases
    Sol_list [0] = False  //
    i = 2
    while i ≤ N :
        j = 1
        while j < i :
            if i % j == 0 :
                if Sol_list [i - j] == False :
                    Sol_list [i] = True
                    break;        // 找到一个就够了
            j += 1
        i += 1

    return Sol_list
```



121. Best Time to Buy and Sell Stock

Say you have an array for which the i^{th} element is the price of a given stock on day *i*.

If you were only permitted to complete at most one transaction (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Note that you cannot sell a stock before you buy one.

Example 1:

```
Input: [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
             Not 7-1 = 6, as selling price needs to be larger than buying price.
```

Example 2:

```
Input: [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

Recurrence Relation:

Base Case:

1、 [ ] && len(Array) == 1 → Output = 0

2、 [a, b] → if b-a > 0 : output = b-a

if b-a <= 0 : output = 0

len(price) > 0 : of size n ，要在在最后一天卖，要么不在

buy_stock (list) = max (max (list [: n-1]) - list[n]),

buy_stock (list [:n-1])

如果是在最后一天卖的，max profit 必然是

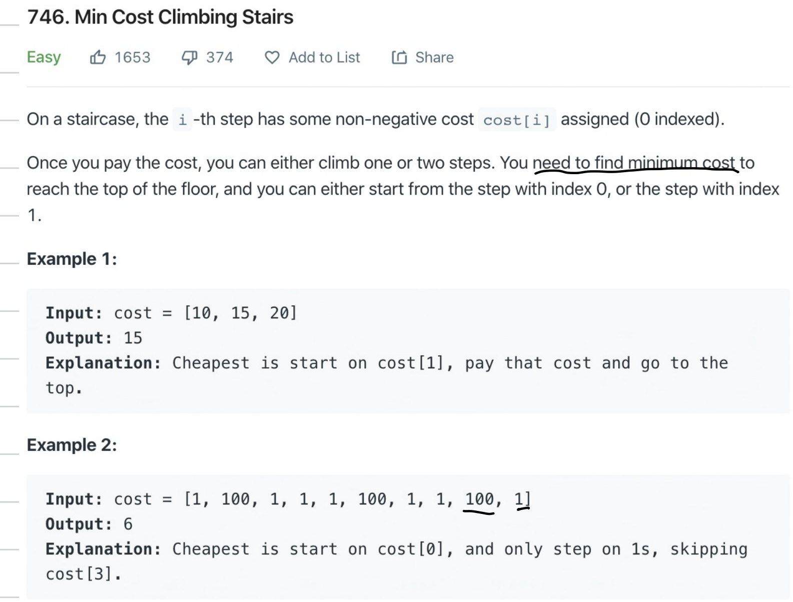
implement :

```
def buy_stock(price):

        sol_list = price[:]

        sol_list[0] = 0
        if price[1] - price[0] < 0
            sol_list[1] = 0
        else :
            sol_list[1] = price[1] - price[0]

        i = 2
        while i < len(price):
            j = 0
            currentMax = price[i] - min(price[: i-1])
            while j < i :
                if sol_list[j] > currentMax:
                    currentMax = sol_list[j]
                j += 1
            sol_list[i] = currentMax
            i += 1
        return sol_list
```



746. Min Cost Climbing Stairs

Easy 👍 1653 👎 374 ♡ Add to List ⎘ Share

On a staircase, the `i` -th step has some non-negative cost `cost[i]` assigned (0 indexed).

Once you pay the cost, you can either climb one or two steps. You need to find minimum cost to reach the top of the floor, and you can either start from the step with index 0, or the step with index 1.

Example 1:

```
Input: cost = [10, 15, 20]
Output: 15
Explanation: Cheapest is start on cost[1], pay that cost and go to the
top.
```

Example 2:

```
Input: cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]
Output: 6
Explanation: Cheapest is start on cost[0], and only step on 1s, skipping
cost[3].
```

相当 Easy

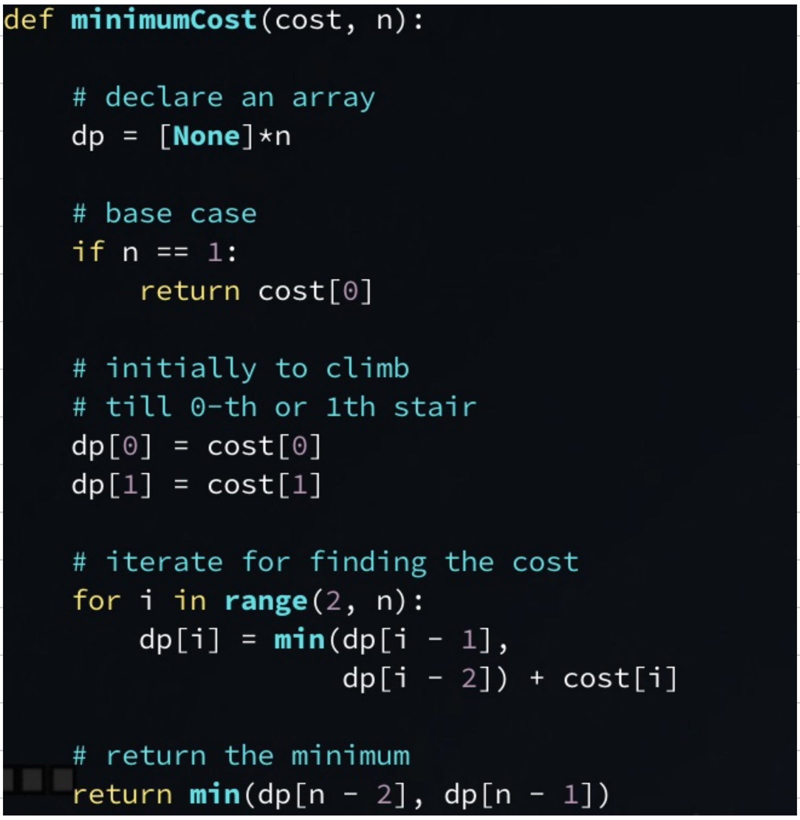
Recurrence Relation :

Base case :

1. [ ] output = 0
2. [a] output = a
3. [a, b] output = min (a, b)

if len (cost) > 2 : of siz n , 要么在这一步给钱，要么在这一步没给

min_stair (cost) = min (cost [n] + min_stair (cost [: n-2]),
min_stair (cost [: n-1]))



```
def minimumCost(cost, n):

    # declare an array
    dp = [None]*n

    # base case
    if n == 1:
        return cost[0]

    # initially to climb
    # till 0-th or 1th stair
    dp[0] = cost[0]
    dp[1] = cost[1]

    # iterate for finding the cost
    for i in range(2, n):
        dp[i] = min(dp[i - 1],
                    dp[i - 2]) + cost[i]

    # return the minimum
    return min(dp[n - 2], dp[n - 1])
```