Solution of Midterm Examination

`Prob1: 16 points (4+4+4+4)`	True or False
`Prob2: 21 points (3+3+3+3+3+6)`	Short-answer questions
`Prob3:  8 points (4+4)`	Making changes
`Prob4: 26 points (3+5+5+3+10)`	Copying Books
`Prob5: 15 points (3+(1+3)+(1+3)+4)`	Revisiting "Closest Pair of Points in 2D"
`Prob6: 20 points (8+6+6)`	Solving the recurrences
`Prob7: 20 points (4+6+4+6)`	Maximum Sub-array, the DP Way
`Prob8:  6 points (6)`	Feedback
`Total:132 points`	

Problem 1. In each of the following question, please specify if the statement is **true** or **false**. If the statement is true, explain why it is true. If it is false, explain what the correct answer is and why. (16 points. For each question, 1 point for the true/false answer and 3 points for the explanations.)

1. The bug report is usually closed by the person who fixes the bug.

   (False) It is usually closed by the person who finds the bug / the tester.

2. The class of NP is the class of languages that cannot be accepted in polynomial time.

   (False)
   The class of NP is the class of languages that can be **verified** in **polynomial time**.
   The class of P is the class of languages that can be **decided** in **polynomial time**.
   The class of P is the class of languages that can be **accepted** in **polynomial time**.

   $\because$ "$P \subseteq NP$" and "languages in P can be **accepted** in **polynomial time**", the description "languages in NP **cannot** be accepted in **polynomial time**" is wrong.

   The term NP comes from **nondeterministic polynomial time** and is derived from an alternative characterization by using **nondeterministic polynomial time** Turing machines. It has nothing to do with "cannot be ... in polynomial time".

3. Different encodings would cause different time complexity for the same algorithm.

   (True)
   The time complexity of the same algorithm are **different** between **unary encoding** and **binary encoding**.
   But if the two encodings are polynomially related (e.g. base 2 & base 3 encodings), then changing between them will not cause the time complexity to change.
   For more information, you can refer to **page 8-13** of the course material "**np_completeness-2_polynomial_time.pdf** ".

4. "A language L is accepted by an algorithm A" means that, for some $x \notin L$, A would reject x.

   (False)
   Please refer to **page 17** of "**np_completeness-2_polynomial_time.pdf** ".
   Even if language L is accepted by an algorithm A, the algorithm will not necessarily reject a string $x \notin L$ provided as input to it. For example, the algorithm may loop forever.

Problem 2. "Short answer" questions: (21 points, 3 points for each question except question 2.6)

1. Why is it necessary for the software tester to minimize the number of steps to generate a bug?

   To make the debugging phase more easy.
   To make it easier to find which step goes wrong.

2. What are the 3 things that need to be in a bug report in a bug tracking system?

   Steps to reproduce,
   What you expected to see, and
   What you saw instead

3. Explain what "polynomially larger" means.

"$f(n)$ is polynomially larger than $g(n)$"
means that "$f(n) = \Omega(n^{\epsilon} \cdot g(n))$ for some $\epsilon > 0$".

Common Mistakes

$f(n)$, $g(n)$ are not necessary to be of the form n^k, nor are they necessary to be polynomials. e.g. $f(n) = 3^n$, $g(n) = 2^n$, then $f(n)$ is indeed polynomially larger than $g(n)$.

Therefore, descriptions like followings are wrong:

「f(n) 和 g(n) 是多項式 ...」

⇒ 不一定會是多項式。f(n) 和 g(n) 的型式並沒有限定。

「f(n) 中 n 的次方項減一個 $\epsilon > 0$ 會比 g(n) 還 ...」

⇒ 可能 f(n) 和 g(n) 根本沒有 n 的次方項

4. Give an example of recurrences that is in the form of $T(n) = aT(\frac{n}{b}) + f(n)$ but cannot be solved with Master theorem.

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

Some interesting recurrences

These are some interesting answers from some students, you can try these yourself.

$T(n) = T(n) + n$

$T(n) = 2T\left(\frac{n}{2}\right) + \cos \pi$ (hint: $\cos \frac{3}{2}\pi = -1$)

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

$T(n) = 3T\left(\frac{n}{4}\right) + log_2 n$

$T(n) = 4T\left(\frac{n}{2}\right) + n^{logn}$

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

5. Explain what would happen if a dynamic programming algorithm is designed to solve a problem that does not have overlapping sub-problems.

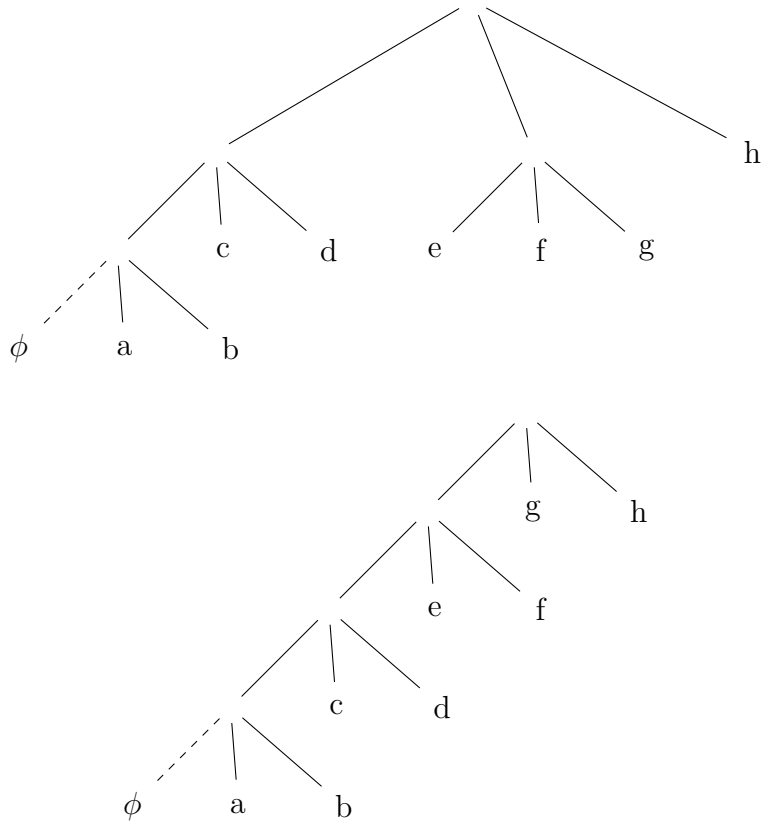
Then it will be just **a waste of memory**, because the answers of sub-problems will never be used again.

And the **running time will be the same** as using D&C algorithm.

6. Derive a ternary Huffman code for the following frequencies of characters: $\{a$:

$100, b : 500, c : 500, d : 600, e : 800, f : 1000, g : 1700, h : 1800\}$. Ternary codes use ternary bits, i.e. $\{0, 1, 2\}$, instead of binary bits, in its codewords. Draw the decoding tree for the code you derive. (6 points)

There are two available answers.



b and c can exchange.

Problem 3. Making changes (8 points)

Assume that there are n kinds of available coins. The denominations are $1 = a_1 < a_2 < \cdots < a_n$ dollars, respectively, and each $a_i, 1 \leq i \leq n$ is an integer. Consider you are making change for m dollars using the *fewest number* of coins.

1. Describe a greedy algorithm to make change when $n = 4$ and the denominations are 1, 5, 10 and 50. Prove that your algorithm always yields an optimal solution (prove the greedy property). (4 points)

Given n denominations $1 \leq a_2 < .. < a_n$ dollars, with a_i mod $a_j == 0$ for each i > j.
Our greedy approach:
always choose the biggest denomination that smaller than the remaining coins.
Prove.. Assume that there exists a optimal solution S such that
it consists of smaller denominations d_1, d_2, .., d_k for a portion of coins
that a bigger denominations d' can fit. And $d_1 + d_2 + .. + d_k = d'$
So in S, we can replace d_1, d_2, .., d_k with d',
then results in smaller solution S' = S - k + 1.
$\Rightarrow$ Contradiction

2. Give a set of denomination for which the greedy algorithm does not yield an optimal solution to some m. Give an example of m to explain. (4 points)

if the set is {1, 6, 10} and the target dollars is 12
greedy will choose {1, 1, 10} $\Rightarrow$ 3 coins
while {6, 6} only need 2 coins to match the request.

Problem 4. Copying Books (26 points)

Before the invention of book-printing, it was very hard to make a copy of a book. All the contents had to be re-written by hand by so called scribers. The scriber had been given a book and after several months he finished its copy. One of the most famous scribers lived in the 15th century and his name was Xaverius Endricus Remius Ontius Xendrianus (Xerox). Anyway, the work was very annoying and boring. And the only way to speed it up was to hire more scribers.

Once upon a time, there was a theater ensemble that wanted to play famous Antique Tragedies. The scripts of these plays were divided into many books and actors needed more copies of them, of course. So they hired many scribers to make copies of these books. Imagine you have m books (numbered $1, 2, \ldots, m$) that may have different number of pages $(p_1, p_2, \ldots, p_m)$ and you want to make one copy of each of them. Your task is to divide these books among k scribes, $k \leq m$. Each book can be assigned to a single scriber only, and every scriber must get a continuous sequence of books. That means, there exists an increasing succession of numbers $0 = b_0 \leq b_1 \leq b_2 \leq \cdots \leq b_{k-1} \leq b_k = m$ such that i-th scriber gets a sequence of books with numbers between $b_{i-1} + 1$ and b_i. The time needed

to make a copy of all the books is determined by the scriber who was assigned the most work. Therefore, our goal is to minimize the maximum number of pages assigned to a single scriber. Your task is to find the optimal assignment.

Answer the following related questions:

1. Define the sub-problem so that we can solve the problem with dynamic programming. Please clearly mark the parameters of the sub-problems and what each parameter represents. (3 points)

   Subproblem(i,j) = Given books from 1 to ith (book$_i$), each corresponding number of pages (book$_i$.pagenum), and j scribers wants to find the minimum number of pages of the maximum number of pages processed by each scriber.

2. In this problem, the cost function that we need to optimize is the maximum number of pages assigned to a single scriber. Using your sub-problem definition, write down the recurrences that determine the value of the cost function based on the ones of smaller related sub-problems. Please also write down the boundary condition(s) of the cost function. (5 points)

   T(i,j) = $min_k(max($T(i-k,j-1), $sum($book$_{k+1\ i}$.pagenum))), i != 1, j != 1
   T(1,j) = book$_1$.pagenum
   T(i,1) = $sum($book$_{1\ i}$.pagenum)

3. (送分) Using a bottom-up style, please write down the pseudo code or the C code of your algorithm to solve the problem in $O(mk)$-time. (5 points)

```
1 Create table dp[m][k]
2 for i in 1 to m do
3   for j in 1 to k do
4     dp[i][j] = 0
5 for i in 1 to m do
6   dp[i][1] = sum(book_{1≤b≤i}.pagenum)
7 for i in 1 to k do
8   dp[1][i] = book_1.pagenum
9 for i in 1 to m do
```

```
10    for j in 1 to k do
11      dp[i][j] = min_{0≤l≤i} (max(T(i-1,j-1), sum(book_{l+1≤b≤i}.pagenum)))
12  answer = dp[m][k]
```

4. (送分) Please analyze the time complexity of your algorithm and show that it is indeed $O(mk)$. (3 points)

Initialization: trivially $O(mk)$
Filling the DP table: three nested for-loop: with respect bounding
$= O(kmj) = O(kmm) = O(m^2k)$

5. Please prove that this problem exhibits optimal substructure. Note that the proof will be in a slightly different format than the ones that we talked about in the lectures. The proof should state that optimal solution(s) to sub-problem(s) can be used to put together the optimal solution to the problem (the big problem). You should start by assuming that you can find the optimal solution(s) to the sub-problem(s), and use it (them) to lead to a solution to the big problem (we usually do it the opposite way). Then, assume that you can find a better solution than this solution to the big problem, and try to find a contradiction which proves that this assumption is false. Then the proof is completed. (10 points)

Assume we can find the optimal solutions to the subproblems,
and use it to lead to a solution S to the big problem.
If there exists some better solution S' of the big problem
such that S' < S. Then in S', there must be some i,
such that book$_{i\leq b\leq k}$ is assigned to a scriber.
So S'= max(optimal_S(i−1,j−1), sum(book$_{i\leq b\leq k}$.pagenum)),
but which cannot be smaller than our solution:
S = $\min_l$(max(optimal_S(i−1,l−1), sum(book$_{l\leq b\leq k}$.pagenum)))
Trivially S ≤ S' ⇒ Contradiction.

Problem 5. Revisiting the Problem of Finding the Closest Pair of Points in 2D Space (15 points)

In the third Divide-and-Conquer lecture, we talked about how to solve the problem with $O(n \log n)$-time. Assume the original set of points is set P. The divide step in the original algorithm presented in the lecture splits P into P_L and P_R and is as the following:

a. Sort the points with their X coordinates. Also sort the points with their Y coordinates. The results will be two sorted sequences of points in P.

b. Let the X coordinate of $\lceil \frac{|P|}{2} \rceil$-th point in the X-sorted sequence be x_c.

c. The first $\lceil \frac{|P|}{2} \rceil$ points in the X-sorted sequence of points in P will be in P_L. And $P_R = P - P_L$.

d. Then the line L that separates P_L and P_R is $x = x_c$. In this case, points both in P_R and P_L are possible to be on line L.

We would now like to change *c.* and *d.* of the divide step to become the following (called the new algorithm in the following):

c. $P_L = \{\forall p \in P, x(p) \leq x_c\}$, and $P_R = P - P_L$. $x(p)$ is the X coordinate of point p.

d. The line L that separates P_L and P_R is still $x = x_c$. In this case, only points in P_L are possible to be on line L.

The conquer step is the same for the original algorithm and the new one with modified divide step:

a. Respectively split the X-sorted and Y-sorted sequences into two sub-sequences according to the membership of points in P_L and P_R.

b. Obtain the closest pairs of points in P_L and P_R, respectively, with recursive function calls. The split sorted sub-sequences are given to the function calls solving the subproblems. Assume the distances between the closest pairs of points in P_L and P_R are δ_L and δ_R, respectively, and $\delta = \min(\delta_L, \delta_R)$.

The following questions are related to the combine steps of both the original algorithm and the new algorithm:

1. In order to keep achieve an $O(n \log n)$-time complexity, what is the maximum time complexity that the combine step can have? Please use the recurrence to explain. (3 points)

   Let the time the combine steps take be $f(n)$, then we expect $2T(\frac{n}{2}) + f(n) = O(nlogn)$.
   By master theorem case 2, $f(n)$ must be $O(n)$ to achieve our goal.

2. In the combine step of the original algorithm, we remove the points in $\{\forall p \in P, |x(p) - x_c| > \delta\}$ in the Y-sorted sequence. Then, we only pair each point to the next N_o points in the sequence and check if the distance between the pair of points is smaller than δ. What is the value of N_o? (1 point) Explain why it is not necessary to check the points after the next N_o points in the sequence. (3 points)

   $N_o = 7$.
   In the region of $\{x_c - \delta \leq x \leq x_c + \delta\}$, we can put at most 8 points without violating the constraint, i.e. for points at the same side, the distance between each two of them is greater then δ. So for each point, we only need to consider at most 7 points that are after it in the y-sorted sequence.

3. With the new algorithm (with the modified divide step) we can follow the same rationale, but now we can instead pair each point to the next N_m points. What is the value of N_m? (1 point) Similarly, please explain the reason in detail. (3 points)

   $N_m = 6$.
   The circumstance is almost the same as the previous question, except that we can only put 3 points at the right side of x_c without violating the constraint since for the points in the right-hand-side, $x_c < p(x) <= x_c + \delta$.

4. The two algorithms have the same time complexity - $O(n \log n)$. In your opinion, which algorithm would run faster in reality? To explain, please list the differences between the two algorithms in the divide step and in the combine step and compare their running time. (4 points)

In reality, the new algorithm would run faster in general.
Divide Step: New one takes $\lfloor \frac{n}{2} \rfloor$ more comparisons than old one does.
Combine Step: New one takes $6n$ comparsisons, and old one takes $7n$.
Compare: Since $\lfloor \frac{n}{2} \rfloor + 6n < 7n$, new one takes less time to finish the whole procedure than old one.

Problem 6. Solving the recurrences (20 points)

Please use the recurrence tree method to solve the recurrence $T(n) = \sqrt{n}T(\sqrt{n}) + n$. The solution needs to be in the Θ-notation, and thus you need to do the following:

1. Draw the recurrence tree and use it to estimate the solution. The solution is in the form of $T(n) = \Theta(f(n))$. (8 points)

   Sum of the i-th level: $\sqrt[2^i]{n} * \prod_{j=1}^{i} \sqrt[2^j]{n} = n$, $i = 0, 1, 2, 3, 4, 5, 6, ...$
   Number of levels: If $2^{2^{k-1}} < n \leq 2^{2^k}$, there would be $\Theta(k) = \Theta(\log(\log(n)))$ levels.
   $T(n) = \Theta(\log(\log(n)))$.

2. Use the substitution method to prove the upper bound. In other words, prove that $T(n) = O(f(n))$. (6 points)

   prove that $T(n) \leq c * n * \log(\log(n))$
   assume that $T(m) \leq c * m \log(\log(m))$ when $m < n$.
   $T(n) = \sqrt{n}T(\sqrt{n}) + n$
   $\leq \sqrt{n} * c * \sqrt{n} \log(\log(\sqrt{n})) + n$
   $= cn \log(\frac{1}{2} \log(n)) + n$
   $= cn(\log(\log(n)) - 1) + n$
   $= cn \log(\log(n)) + (1 - c)n$
   $\leq cn \log(\log(n))$ if $c \geq 1$.
   boundary cases:
   assume that $T(4) = k$ where k is a constant, $k \leq 4c$ when $c \geq \frac{k}{4}$.
   $\therefore T(n) = O(cn \log(\log(n)))$.

3. Use the substitution method to prove the lower bound. In other words, prove that $T(n) = \Omega(f(n))$. (6 points)

prove that $T(n) \geq c * n * \log(\log(n))$

assume that $T(m) \geq c * m \log(\log(m))$ when $m < n$.

$T(n) = \sqrt{n}T(\sqrt{n}) + n$

$\geq \sqrt{n} * c * \sqrt{n} \log(\log(\sqrt{n})) + n$

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

$= cn(\log(\log(n)) - 1) + n$

$= cn \log(\log(n)) + (1 - c)n$

$\geq cn \log(\log(n))$ if $c \leq 1$.

boundary cases:

assume that $T(2) = k$ where k is a constant, $k \geq 2c \log(\log(2)) = 0$ when $c > 0$.

$\therefore T(n) = \Omega(cn \log(\log(n)))$.

Problem 7. Maximum Sub-array, the Dynamic Programming Way (20 points)

In the Divide-and-Conquer lectures, we talked about how to derive an $O(n \log n)$-time algorithm to solve the problem. As we stated in the lecture, this is actually not the fastest algorithm to solve the problem. Dynamic programming actually can be used to solve the problem in linear time. Let's re-state the problem as follows.

You are given a sequence of n numbers $\{a_1, a_2, \ldots, a_n\}$ in an array. You are asked to find the sum of the maximum sub-array. A sub-array is given by $\{a_i, a_{i+1}, \ldots, a_j\}, 1 \leq i, j \leq n$. Note that 0, the sum of an empty sub-array, i.e., $j < i$, is allowed to be given as the answer. For example, the case could happen when all values in the array are negative.

To give you easier time, we will define the sub-problem for you, given as the following. The sub-problem $S_i, 1 \leq i \leq n$, is to defined as the one to find the maximum sub-array that starts from anywhere but *ends on* a_i.

1. Prove that, with the given definition, the problem exhibits optimal substructure. (4 points)

Denote A_i as the optimal solution of S_i, the maximum sub-array that ends on a_i, and V_i as the sum of A_i. Note that A_i can be an empty array, so $V_i \geq 0$,
Assume a_{i-1} in A_i. There exists a V_{i-1} such that $V_i = V'_{i-1} + a_i$.
If $V'_{i-1} < V_{i-1}$, $V_i = V'_{i-1} + a_i < V_{i-1} + a_i$,
which is a contradiction because V_i should be the maximum.

2. Use the recurrences to give the cost function that represents the sum of the maximum sub-array of the given sub-problem. Note that in different conditions, the function is given in different ways. And you also need to consider and give the boundary cases. (6 points)

$$c[i] = \begin{cases} \max(a_1, 0) & \text{if } i = 1; \\ \max(c[i-1] + a_i, 0) & \text{if } i > 1. \end{cases}$$

3. Assume that you are given an array as the following: $\{1, -4, 3, 2, -1, 3, 5, -4\}$. As an example, fill out the cost function table to solve all sub-problems for the given array. Then, explain and show how you can use the results to obtain the solution of the original problem (maximum sub-array of the original array). (4 points)

i	1	2	3	4	5	6	7	8
$c[i]$	1	0	3	5	4	7	12	8

The solution of the original problem is the maximum $c[i]$.
In this example, the solution is 12.

4. Instead of outputting the maximum value of the sum of the sub-array, you are now asked to give the indices of the starting and ending elements of the maximum sub-array. Please write down the pseudo code or C code to output these two numbers. (6 points)

```
1 max_sum = 0, sum = 0
2 start = 2, end = 1, temp_start = 1
3 for i = 1 to n
4     sum += a_i
5     if (sum < 0) {
6         sum = 0
```

```
7           temp_start = i+1
8       }
9       if (sum > max_sum) {
10          max_sum = sum
11          start = temp_start
12          end = i
13      }
14  output (start, end)
```

Problem 8. I have the tradition of letting the students write some feedbacks about the course in the exam and I would like to continue this tradition. Please write down 3 things you like about this course and 3 things that you would like to see some changes (and your suggestion about how we should change them). (6 points)