Algorithms and Data Structures
Final Review

Keyi Yuan
Teaching assistant

Dec.23th 2019

Agenda

- Recent Deadlines:
  - Weekly Homework 13: Dec.27th, 23:59 (Thursday)
  - Programming Homework 4: Dec.31st, 23:59 (Tuesday)

Agenda

- Exam Time:
  - Jan.9th, 10:00-12:00, 120 minutes in total
  - The time of regrading may be very tight! We will inform you.
- Covers:
  - The lectures from *Lecture 15 Shortest Path* to *Lecture 26 NP-Complete*
  - The discussions from *week 10* to *week 16.*
  - But not all content will be covered. See more details on Piazza.
- Policy:
  - Closed book. No cheating sheet.
  - No electronic devices. For example, calculators, PC, smartphones and so on.
  - And any other actions that violate the rule of the exam will be judged as plagiarism and zero score immediately.

Agenda

- Covers:
  - 4 Multiple Choices (Choose all correct answers, which worth 20 points): You should notice that if you choose the subset of the correct solution except choosing nothing can get the half point of this question.
  - 7 General Problems (which worth 80 points).
- OH may be adjusted. Please also see it on Piazza.
  - We will end our OH before the end of Week 16 normally.

Claim

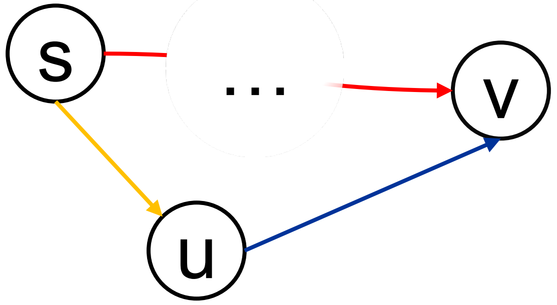
1. Topics that reviewed in this discussion may not be covered in the Exam
2. Topics that not reviewed in this discussion may be covered in the Exam
3.

③ As we all know that Statements start with "Keyi yuan thinks" usually won't be true. (X

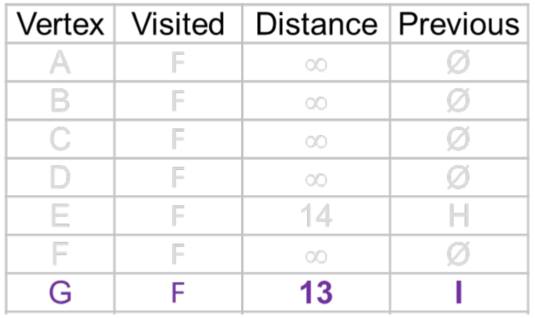
1. Shortest Path

SSSP: Dijkstra's Algorithm

- It is a Greedy Algorithm. Because the relaxation of a vertex can be done before it is visited. After it is visited, we will never relax it. We visit the vertex one by one.
- Relaxation: $d(s, v) \leftarrow \min(d(s, v),\ d(s, u) + w(u, v))$
- It can be used in both undirected and directed graph.
- It can not be used in a graph with negative weight edge or negative cycle.
- Space complexity: $\Theta(|V|)$. Because of initialization.



```
RELAX(u, v, w)
1  if $v.d > u.d + w(u, v)$
2      $v.d = u.d + w(u, v)$
3      $v.\pi = u$
```



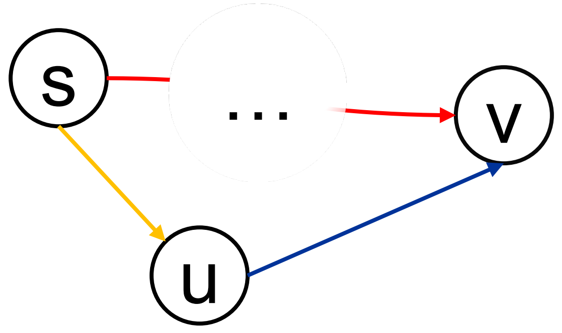
Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	14	H
F	F	∞	Ø
G	F	13	I

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN(Q)
6      $S = S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX(u, v, w)
```

- You should know the process of how to find the shortest path.
  - Recommend you to find an example to realize it by hand.
  - Notice when to mark a vertex as "visited".

SSSP: Dijkstra's Algorithm

- Each time we find a vertex, we must check all of its neighbors
  - With an adjacency matrix, the run time is
    - $\Theta(\ |V|(|V|(\text{find closest}) + |V|(\text{relaxation}))\ ) = \Theta(|V|^2)$
  - With an adjacency list, the run time is
    - $\Theta(\ |V|^2\ (\text{find closest}) + |E|(\text{relaxation})\ ) = \Theta(|V|^2)$ as $|E| = O(|V|^2)$
  - Using a binary heap, the time complexity can be reduced to $O((|V|\ +\ |E|)\log|V|)$.
    - $O(|V|\ \ln\ \ (|V|))$: find closet vertex and pop it(recall time complexity for a binary heap)
    - $O(|E|\ \ln\ \ (|V|)))$: for each edge, we do a relaxation, which influences the estimated distance for a vertex. We should maintain the heap.



```
RELAX(u, v, w)
1  if $v.d > u.d + w(u, v)$
2      $v.d = u.d + w(u, v)$
3      $v.\pi = u$
```

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN(Q)
6      $S = S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX(u, v, w)
```

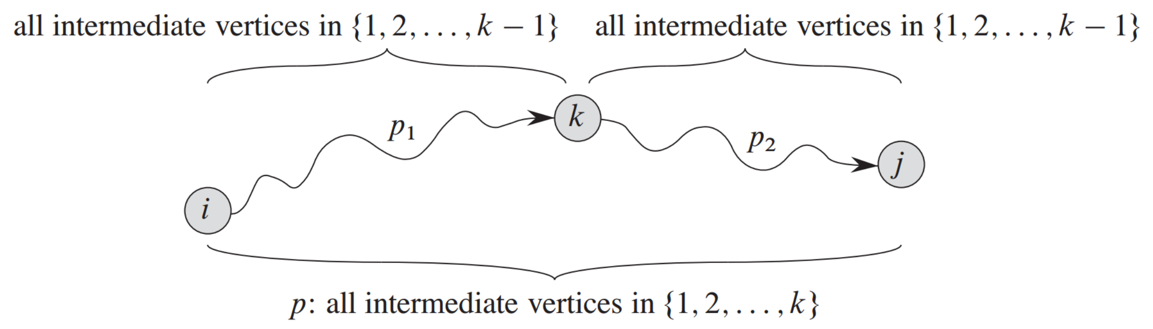
APSP: Floyd-Warshall Algorithm

- It is a Dynamic Programming algorithm. It breaks the problem down into smaller subproblems, then combines the answers to those subproblems to solve the big, initial problem.
- The bellman equation: $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
- It can be used in negative weight edges with no negative weight cycle.
- Initialization:

$$d_{i,j}^{(0)} = \begin{cases} 0 & \text{If } i = j \\ w_{i,j} & \text{If there is an edge from } i \text{ to } j \\ \infty & \text{Otherwise} \end{cases} \qquad p_{i,j} = \begin{cases} \varnothing & \text{If } i = j \\ j & \text{If there is an edge from } i \text{ to } j \\ \varnothing & \text{Otherwise} \end{cases}$$

- Instead of running Dijkstra's algorithm n times, it takes O(n^3) time overall.

```
for ( int k = 0; k < num_vertices; ++k )
    for ( int i = 0; i < num_vertices; ++i )
        for ( int j = 0; j < num_vertices; ++j )
            if ( d[i][j] > d[i][k] + d[k][j] )
                p[i][j] = p[i][k];
                d[i][j] = d[i][k] + d[k][j];
```


all intermediate vertices in $\{1, 2, \ldots, k-1\}$
all intermediate vertices in $\{1, 2, \ldots, k-1\}$
p_1
k
p_2
i
j
p: all intermediate vertices in $\{1, 2, \ldots, k\}$

Ex. Decrease the shortest path

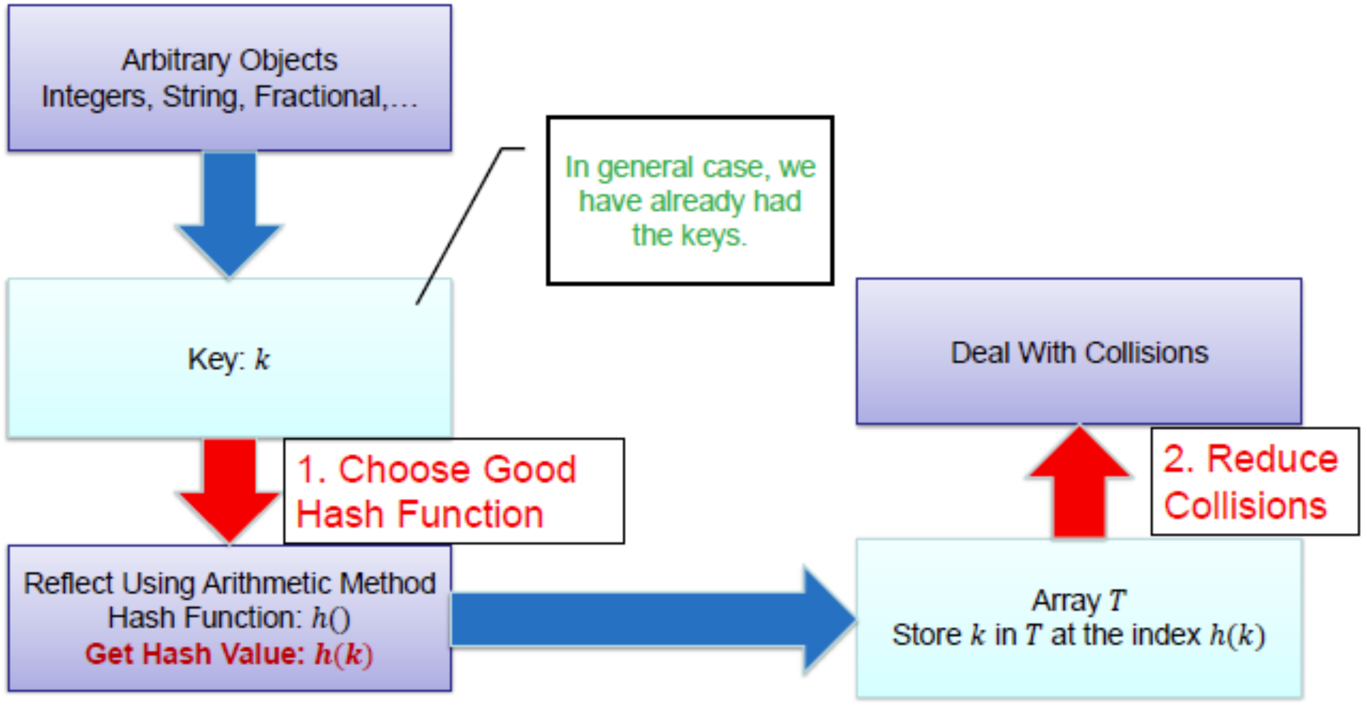
- We have a undirected graph $G = (V, E)$ and use Dijkstra algorithm we can get the shortest path between s and t. We have another undirected graph $G' = (V, F)$. We want to add one edge e' to this graph and $e' \in F$. We want this addition would result in the maximum decrease of the shortest distance between s and t. Since the set F of the edges can be quite large running Dijkstra F times is too slow. You should design a more clever algorithm.

- Create 2 copies of the graph G, say G_A and G_B. First, convert all edges in both of these graphs to be bi-directed from undirected. Next, for each edge $e: (u, v) \in F$: add an edge from u in G_A to v in G_B. Also, add an edge from v in G_A to u in G_B. Now run Dijkstra's starting from s in G_A, and find the distance to t in G_B. If this distance is longer than the original distance from s to t (checked using another run of Dijkstra's), then return None.
- The correctness for this algorithm in guaranteed by the fact that we end in G_B, and the only way to get to G_B is by using one and only one edge in F. The runtime is going to be that of a single run of Dijkstra's on a graph with $2|V|$ vertices, and $2|E| + |F|$ edges, which is $O((2|V| + 2|E| + |F|) \log 2|V|) = O((|V| + |E| + |F|) \log |V|)$

2. Hash Table

Hash Table Overview

- Hash Table is a data structure which can support Insert(), Delete() and Search() all in an expectation time $O(1)$. (Assume hash function takes $O(1)$ time to compute).
  - But the worst time complexity for each operation is still $O(n)$.
  - Motivation: Use less space, reduce time complexity.
  - Main idea: Convert key to much smaller value, and store at this location.


Arbitrary Objects
Integers, String, Fractional,…
In general case, we have already had the keys.
Key: k
1. Choose Good Hash Function
Reflect Using Arithmetic Method
Hash Function: $h()$
Get Hash Value: $h(k)$
Array T
Store k in T at the index $h(k)$
2. Reduce Collisions
Deal With Collisions

Hash Table Part 1: Hash function

- We will use a hash function to determine where the key should be stored. It should be stored at index $h(k)$.
  - Division method: $h(k)$ = k mod m
  - Multiplication method: $h(k) = \lfloor m\ (k\ A \bmod 1) \rfloor$, where A is some constant.
- Hash function should have four properties:
  - Should be fast: ideally $\Theta(1)$
  - The hash value must be deterministic
    - $h(k)$ always has the same value.
  - Equal objects hash to equal values
    - $x = y \Rightarrow h(x) = h(y)$
  - If two objects are randomly chosen, there should be only a $1/m$ chance that they have the same hash value.

Hash Table Part 2: Collisions

- Method 1: Closed addressing -> Chaining method.
- For different keys, if their hash values are same, they should be inserted into a linked list.
- Load factor: $\alpha = n/m$, where n is the number of inserted keys and m is the number of slots.
    - Under uniform hashing assumption, the average(expected) time for each operation is $O(\alpha)$.
    - However, even with uniform hashing, the worst case performance is $O(n)$.
- Advantage: Easy to insert and delete.
- Disadvantage: Use more space.

Hash Table Part 2: Collisions

- Method 2: Open addressing: each hash table entry stores at most one key.
- Probe sequence: $h(k,0), h(k,1), \ldots, h(k,m-1)$, which determines where keys store.
  - If $h(k,0), \ldots, h(k,i)$ already occupied, try to insert location in $h(k,i+1)$. Otherwise, stop.
- Linear probing: $h(k,i) = (h'(k) + i) \bmod m$, where h' is an ordinary hash function.
  - Poor performance due to primary clustering, i.e., with more insertions, the time will be longer.
- Quadratic probing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where c_1, c_2 are constants.
  - No primary clustering, but secondary clustering, where $h(k_1,0) = h(k_2,0)$ implies both probe sequences are equal.
- Advantage: Save more space. Quadratic causes no more collisions than that of linear.
- Disadvantage: Delete is more complicated. See it more in discussion session.

3. Sorting

Sort Overview

- Merge/Quick Sort uses the method of Divide and Conquer.
- Specially, for quick sort, the average case is much more often.

	Average	**Worst**	**Additional Space**
Insert Sort	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Heap Sort	$\Theta(n \log n)$	$O(n \log n)$	$O(1)$
Merge Sort	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$

Merge Sort

- Algorithm:
  - Divide: Partition n elements array into two sub lists with $n/2$ elements each. Partition until the length of sub lists is 1.
  - Conquer: Sort sub list1 and sub list2 by merging.
  - Combine: Merge sub list1 and sub list2 at last.
- The additional space is $O(n)$ because you need to store the sorted sub-list.
- You should know the process of the sorting.
- It can be used to count inversions.

```
void mergesort(n) {
  if (n==1)
    return;
  else {
    L=mergesort(n/2);
    R=mergesort(n/2);
  }
  // takes O(n) time
  merge(R,L);
}
```

Quick Sort

- Algorithm:
  - Divide: Partition (rearrange) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1..r]$. Compute the index q as part of this partitioning procedure.
  - Conquer: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.
  - Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.
- The additional space is $O(\log n)$ because of system recursive calls.
- In discussion, lectures, there are many ways to choose a pivot.
- But you should understand that for each steps what you need to do and why the worst case is $\mathrm{O}(n^2)$.

4. Divide and Conquer

Divide and Conquer

- Divide a size n problem into a $(a \geq 1)$ sub-problems with size $\frac{n}{b}$.
- Template of Divide & Conquer
  - Step 1: Divide the problem into sub-problems (Divide)
  - Step 2: Solve each sub-problem (Conquer)
  - Step 3: Combine the results of sub-problems. (Combine)
- The example you have learned:
  - Merge Sort
  - Quick Sort
  - Median of Medians
  - Integer Multiplication
  - Matrix Multiplication
  - Closest Pairs of Points
  - Maximum Subarray.

Divide and Conquer

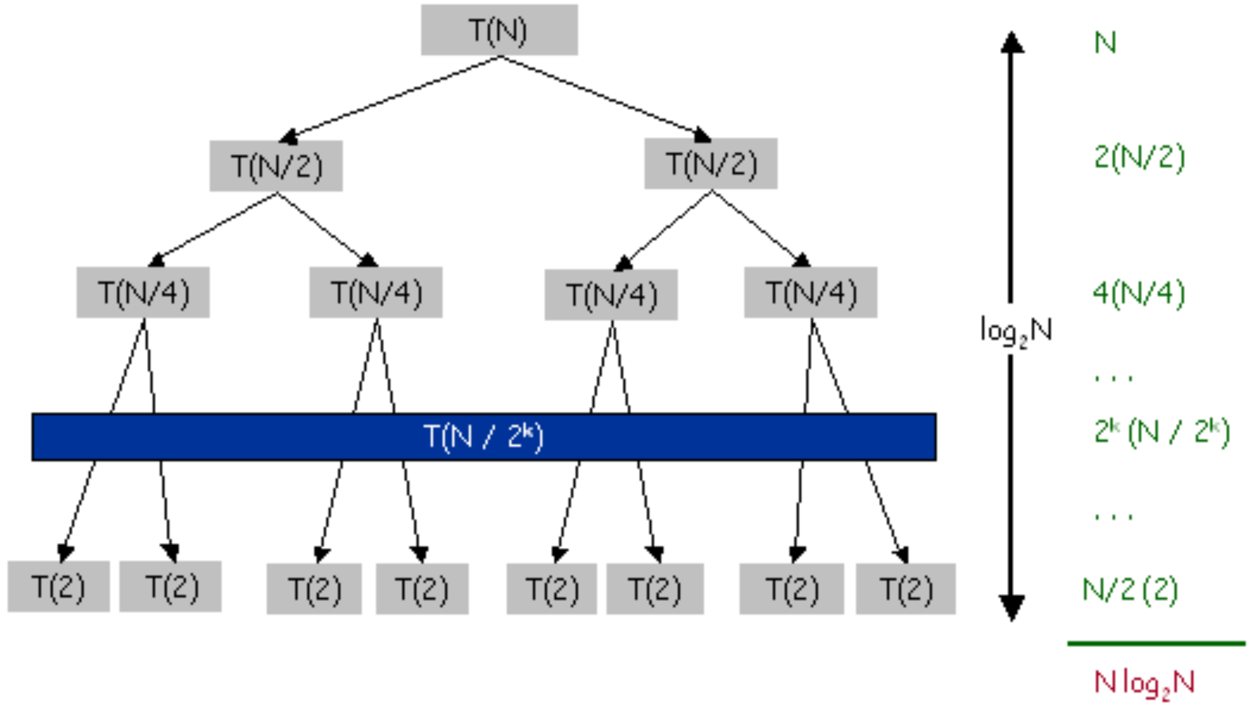
- When to use Divide and Conquer? If you have found:
  - The problem can be easily solved if the scale of the problem is reduced to a certain extent.
  - The problem can be decomposed into several same problems of a smaller scale, that is, the problem has the optimal substructure property.
  - The solutions of the sub-problems decomposed by this problem can be merged into the solutions of this problem.
  - The sub-problems decomposed by this problem are independent of each other, that is, there are no common sub-problems among them.
- The key point:
  - The keyword: “Divide and Conquer”, “divide”, “merge”, “recursive”, ……
  - The recursive function: 4 methods to solve it.
    - (1) Induction method (Guess, Prove) (2) Substitution method [See them in previous discussion]
    - (3) Recursion Tree (4) Master Theorem

Solve Recursive Function: Recursion Tree

- Visualize the recursive calls that occur during mergesort(n).
- There are $\log_2 n$ levels in the recursion tree.
- At level i, there are 2^i recursive calls mergesort($n/2^i$).
- Each call does $n/2^i$ work in merge function.
    - So total work at level i is $2^i * \left(\frac{n}{2^i}\right) = n$.
- So total work overall is $S(n) = n \log_2 n$.

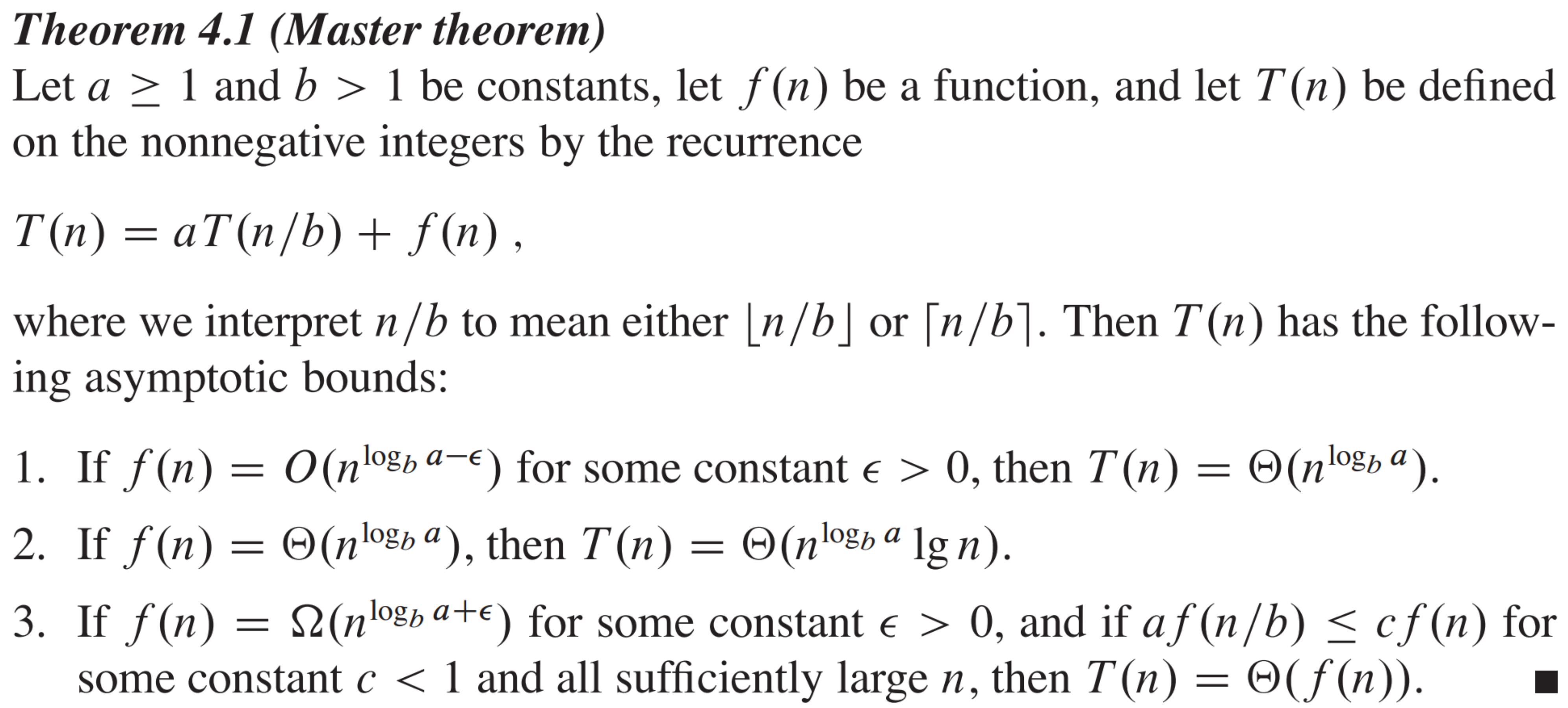
```
void mergesort(n) {
  if (n==1)
    return;
  else {
    L=mergesort(n/2);
    R=mergesort(n/2);
  }
  // takes O(n) time
  merge(R,L);
}
```

- (1) Calculate how many levels.
- (2) Calculate how many recursive calls each level has.
- (3) Calculate how many work does each recursive call does.
    - Then calculate the total work on each level.
- (4) Calculate how many total work you need to do.



Solve Recursive Function: Master Theorem

- Based on comparing the nonrecursive complexity $f(n)$ with $n^{\log_b a}$.



Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Solve Recursive Function: Master Theorem

- Ex $T(n) = 9T\left(\frac{n}{3}\right) + n$
  - $a = 9, b = 3, f(n) = n, \log_b a = 2.$
  - Check $f(n) = O(n^{2-\epsilon})$, so use case 1 of Master theorem.
  - So $T(n) = \Theta(n^2)$.
- Ex $T(n) = T\left(\frac{2n}{3}\right) + 1.$
  - $a = 1, b = \frac{3}{2}, f(n) = 1, \log_b a = 0.$
  - $f(n) = \Theta(n^0)$, so use case 2 of theorem.
  - So $T(n) = n^0 \log n = \Theta(\log n)$.
- Ex $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$.
  - $a = 3, b = 4, f(n) = n \log n, \log_b a \approx 0.793.$
  - $f(n) = \Omega(n^{0.793+\epsilon})$, so use case 3 of theorem.
  - So $T(n) = \Theta(n \log n)$.

Solve Recursive Function: Master Theorem

- Note in cases 1 and 3, $f(n)$ needs to be smaller (resp. larger) than $n^{\log_b a}$ by a polynomial factor n^{ϵ}.
  - If this doesn't hold, we can't use the theorem.
- Ex $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$.
  - $a = 2, b = 2, f(n) = n \log n\,, \log_b a = 1$.
  - However, case 2 of the Master theorem doesn't apply, since $f(n) \neq \Theta(n)$.
  - Case 3 also doesn't apply, since $n \log n \neq \Theta(n^{1+\epsilon})$ for any $\epsilon > 0$.
  - So we can't use the Master theorem to solve this recurrence.

5. Greedy Algorithm

Greedy Algorithm

- A greedy algorithm is an algorithm that constructs an object X one step at a time, at each step choosing the locally best option.
- In most of cases, greedy algorithms construct the globally best object by repeatedly choosing the locally best option.
- How to define best?
- The example you have learned:
  - Coin changing
  - Interval scheduling
  - Scheduling to minimizing lateness
  - Huffman coding
  - Dijkstra’s Algorithm (Shortest Path)
  - Prim’s, Kruskal’s Algorithm (Minimum Spanning Tree)
  - Single-link clustering.

Greedy Algorithm

- The key point:
  - The keyword: “Greedy Algorithm”, “best”, “sort”,, ……
  - Analyze time complexity by pseudocode.
- How to prove correctness? Hard!
  - Greedy algorithms are often used to solve optimization problems: you want to maximize or minimize some quantity subject to a set of constraints.
  - When you are trying to write a proof that shows that a greedy algorithm is correct, you often need to show two different results.
  - First, you need to show that your algorithm produces a feasible solution, a solution to the problem that obeys the constraints.
  - Next, you need to show that your algorithm produces an optimal solution, a solution that maximizes or minimizes the appropriate quantity.

Greedy Algorithm

- How to write the format of Greedy Algorithm? “Greedy stays ahead”
- **Define Your Solution**. Your algorithm will produce some object X and you will probably compare it against some optimal solution X^*. Introduce some variables denoting your algorithm's solution and the optimal solution.
- **Define Your Measure**. Your goal is to find a series of measurements you can make of your solution and the optimal solution. Define some series of measures $m_1(X), m_2(X), \ldots, m_n(X)$ such that $m_1(X^*), m_2(X^*), \ldots, m_k(X^*)$ is also defined for some choices of m and n. Note that there might be a different number of measures for X and X^*, since you can't assume at this point that X is optimal.
- **Prove Greedy Stays Ahead**. Prove that $m_i(X) \geq m_i(X^*)$ or that $m_i(X) \leq m_i(X^*)$, whichever is appropriate, for all reasonable values of i. This argument is usually done inductively.
- **Prove Optimality**. Using the fact that greedy stays ahead, prove that the greedy algorithm must produce an optimal solution. This argument is often done by contradiction by assuming the greedy solution isn't optimal and using the fact that greedy stays ahead to derive a contradiction.

6. Dynamic Programming

Dynamic Programming

- Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions using a memory-based data structure.
- Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup. So the next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time. This technique of storing solutions to subproblems instead of recomputing them is called memoization.
- The example you have learned:
    - Weighted interval scheduling
    - Segmented least squares
    - Knapsack problem
    - Stairs climbing problem

Dynamic Programming

- The key point:
  - The keyword: “Dynamic Programming”, “subproblem”, “look-up”, ”Bellman equation”, ……
  - Analyze time complexity by pseudocode.
- The general outline of a correctness proof for a dynamic programming algorithm is as following:
- **Define Subproblems.** Dynamic programming algorithms usually involve a recurrence involving some quantity OPT($k_1, \dots, k_n$) over one or more variables Define what this quantity represents and what the parameters mean.
- This might take the form:
  - “OPT(k) is the maximum number of people that can be covered by the first k cell towers” or
  - “OPT(u, v, i) is the length of the shortest path from u to v of length at most i.”
- Form like: If in case 1, then … is the best. Otherwise in case 2 … is the best. They form OPT(k).

Dynamic Programming

- **Write a Recurrence**. Now that you've defined your subproblems, you will need to write out a recurrence relation that defines OPT($k_1, \ldots, k_n$) in terms of some number of subproblems. Make sure that when you do this you include your base cases.
- **Prove the Recurrence is Correct**. Having written out your recurrence, you will need to prove it is correct. Typically, you would do so by going case-by-case and proving that each case is correct. In doing so, you will often use a “cut-and-paste” argument to show why the cases are correct.
- **Prove the Algorithm Evaluates the Recurrence**. Next, show that your algorithm actually evaluates the recurrence by showing that the table values match the value of OPT and that as you fill in the table, you never refer to a value that hasn't been computed yet. To be fully rigorous, you would probably need to prove this by induction. For the purposes of CS101, you do not need to go into this level of detail. A few sentences should suffice here.

Dynamic Programming

- **Prove the Algorithm is Correct.** Having shown that you've just evaluated the recurrence correctly, your algorithm will probably conclude with something like “return A[m, n]” or “return C[0].” Prove that this table value is the one that you actually want to read.
- This might seem like a lot to do, but most of these steps are very short. The main challenge will be proving that the recurrence is correct, and most of the remaining steps are there either to set up this step or to show how that step leads to correctness.

7. Reduction, P/NP and NP-complete

Decision Problems

- Def: assignment of inputs to No(0) or Yes(1)
- Inputs are either **No instances** or **Yes instances** (i.e. satisfying instances)

Problem	Decision
Independent Set	Given a graph, does it contains a subset of k (or more) vertices such that no two are adjacent?
3-SAT	Given a CNF formula, dost it have a satisfying truth assignment?
Hamilton Cycle	Given an undirected graph, does there exist a cycle that visits
3-coloring	Given an undirected graph, can the nodes be colored black, white, and blue so that no adjacent nodes have the same color?

Reduction Review

- Suppose you want to solve problem A.
- One way to solve is to convert A into a problem B you know how to solve.
- Solve using an algorithm for B and use it to compute solution to A.
- This is called a reduction from problem A to problem B $(A \to B)$
- Because B can be used to solve A, B is **at least as hard** $(A \leq B)$
- If you can solve B, you can also solve A.

P and NP

- **P:** the set of decision problems for which there is an algorithm A such that for every instance I of size n, A on I runs in poly(n) time and solves I correctly.
- **NP:** the set of decision problems for which there is an algorithm V , a "certifier", that takes as input an instance I of the problem, and a "certificate" bit string of length polynomial in the size of I, so that:
  - V always runs in polynomial time, and the input and output should be in the polynomial size of I
  - if I is a YES-instance, then there is some certificate c so that V on input (I, c) returns **YES**, and
  - if I is a NO-instance, then no matter what c is given to V together with I, V will always output **NO** on (I, c).
- **For NP Problem, an instance can be verified in polynomial time.**
- You can think of the certificate as a proof that I is a YES-instance. If I is actually a NO-instance then no proof should work.

All problems in P are in NP

- Notice that for problems in P, V doesn’t need a certificate y.
  - For problems in P, it’s easy to determine if they’re solvable or not by itself.
  - $P \subseteq NP$

- In fact, a polytime verifier isn’t powerful enough to find a nontrivial factor of an input.
  - But if it’s given a nontrivial factor, it can check the factor works in polytime, and therefore verify the input is composite.

P vs NP

- NP != Not Polynomial Time Problem
- NP = Nondeterministic Polynomial Time Problem

- P = NP? P ≠ NP? This is an open question.
- Why do we care? If can show a problem is hardest problem in NP, then problem cannot be solved in polynomial time if P ≠ NP.

- How to define hardest? How do we relate difficulty of problems? Reductions!

NP-completeness

- Out of all the NP problems, there's a subset of NP problems called NP-complete (NPC) problems that are the "hardest" NP problems.

- To determine whether P=NP, it suffices to know whether P=NPC.
  - If the hardest problems can be solved in polytime, then all NP problems can be solved in polytime. i.e. P=NP.

- So the study of P vs NP focuses on NPC problems.

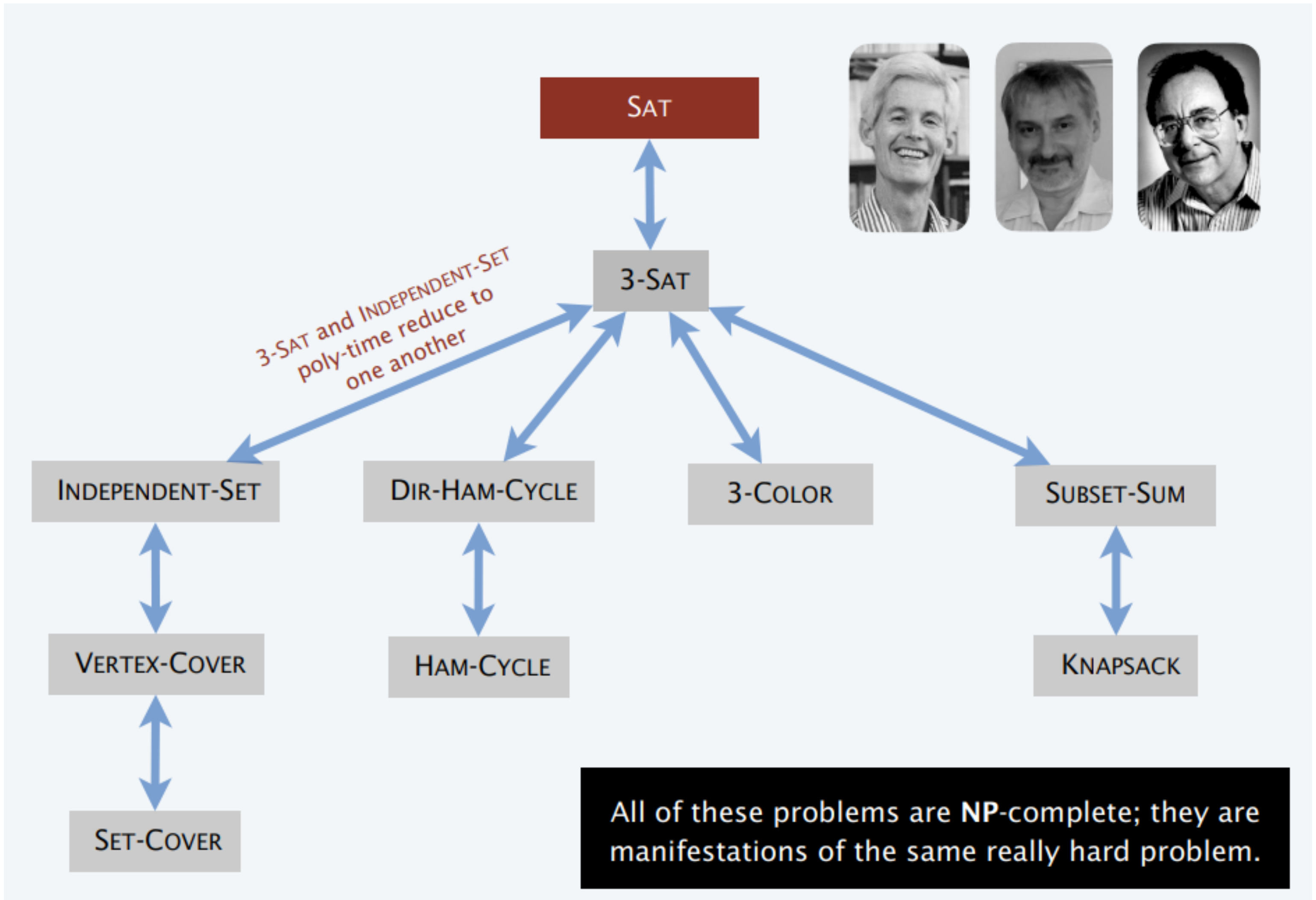
NP-completeness

- Def A problem A is NP-complete (NPC) if the following are true.
  - $A \in NP$.
  - Given any other problem $B \in NP$, $B \leq_P A$.

- Thus, a NP-complete problem is an NP problem that can be used to solve any other NP problem.
  - It's a "hardest" NP problem.

The web of NP-completeness

- You can use it in exam directly.


SAT
3-SAT
3-SAT and INDEPENDENT-SET poly-time reduce to one another
INDEPENDENT-SET
DIR-HAM-CYCLE
3-COLOR
SUBSET-SUM
VERTEX-COVER
HAM-CYCLE
KNAPSACK
SET-COVER
All of these problems are NP-complete; they are manifestations of the same really hard problem.

How to prove NPC?

- Thm 1 Given two NP problems A and B, suppose A is NP-complete, and $A \leq_P B$. Then B is also NP-complete.
- Proof Let C be any NP problem. Then $C \leq_P A$, since A is NP-complete.
  - Since $A \leq_P B$, then by Theorem 1, we have $C \leq_P A \leq_P B$.
  - Since also $B \in NP$, then B is NPC.

- To prove a problem B is NP-complete
  - First, you have to prove $B \in NP$, but that's usually not hard.
  - Second, take a problem A you know is NPC, and prove $A \leq_P B$.
    - To prove $A \leq_P B$, you need to give a polytime reduction from A to B.
    - $\Rightarrow$: To say B is harder than A, which has been known as a NPC problem.
    - $\Leftarrow$: Because B is NPC as well, A and B are **"equivalently" hard**. This means B can also be reduced to A.

NP-completeness and P vs NP

- Thm 2 Suppose a problem A is NP-complete, and $A \in P$. Then P=NP.
- Proof Consider any other NP problem B. We'll show $B \in P$.
  - Since A is NPC, there's a polytime mapping f from B to A.
  - Given an instance X of B, run f on X to get an instance Y of A.
  - Since $A \in P$, there's a polytime algorithm g to solve A.
  - Run $g(Y)$, and return the same answer for X.
  - By the definition of $\leq_P$, $g(Y)$ is true $\Leftrightarrow$ X is true.
  - Running f and g both take polytime. So we can solve B in polytime.

NP-completeness and P vs NP

- Cor Suppose a problem A is NP-complete, and $A \notin P$. Then for any NP-complete problem B, $B \notin P$.
  - If $B \in P$, then since B is NPC, we have P = NP by Theorem 2. So since $A \in NP$, we have $A \in P$, a contradiction.

- To prove $P \neq NP$ (which is what most people think), it's enough to show one NPC problem is not solvable in polytime, by the corollary.
  - Nor has anyone shown a polytime algorithm for any NPC problem.

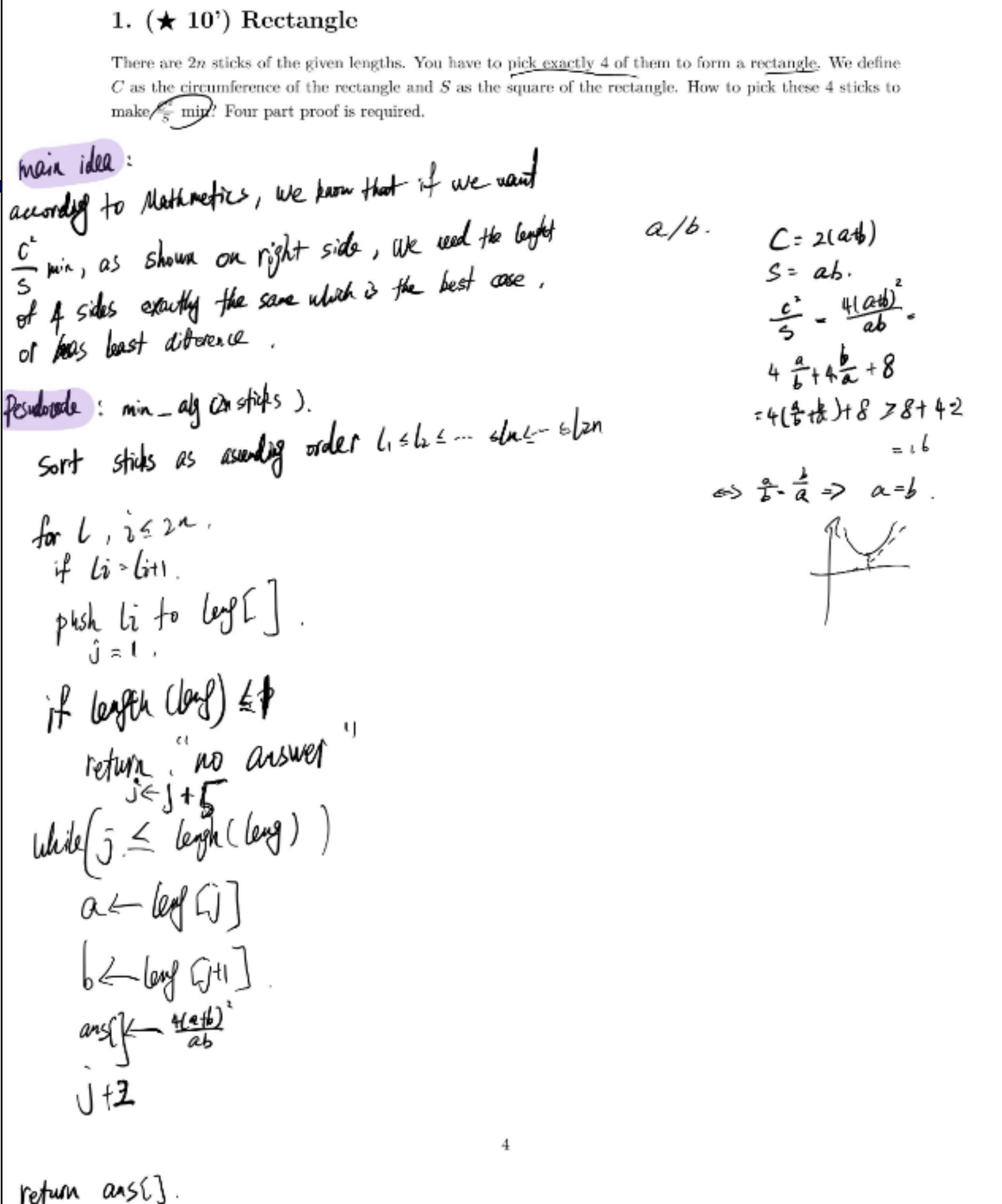
8. What do you need care about in exam?

1. Additional Points

- We will set additional 2 points in the final exam.
  - If you do this well, you won’t get any score.
  - But if you violate the following rules, your score will be **deducted**.
- 2 points: You should write down your name and your student ID on the top of **each** page of the answer sheet.

1. Clarity Example 1

- The code part and main idea part look like a mass…
- The font is not clear!
- We will judge unclear answer as wrong answer!



1. (★ 10') Rectangle

There are $2n$ sticks of the given lengths. You have to pick exactly 4 of them to form a rectangle. We define C as the circumference of the rectangle and S as the square of the rectangle. How to pick these 4 sticks to make $\frac{C^2}{S}$ min? Four part proof is required.

main idea :
according to Mathmetics, we know that if we want $\frac{C^2}{S}$ min, as shown on right side, we need the length of 4 sides exactly the same which is the best case, or ~~has~~ least diference.

a/b.

$C = 2(a+b)$
$S = ab$.
$\frac{C^2}{S} = \frac{4(a+b)^2}{ab}$.
$4\frac{a}{b} + 4\frac{b}{a} + 8$
$= 4(\frac{a}{b} + \frac{b}{a}) + 8 \geq 8 + 4 \cdot 2$
$= 16$
$\Leftrightarrow \frac{a}{b} = \frac{b}{a} \Rightarrow a = b$.

Pesudocode : min_alg (n sticks).

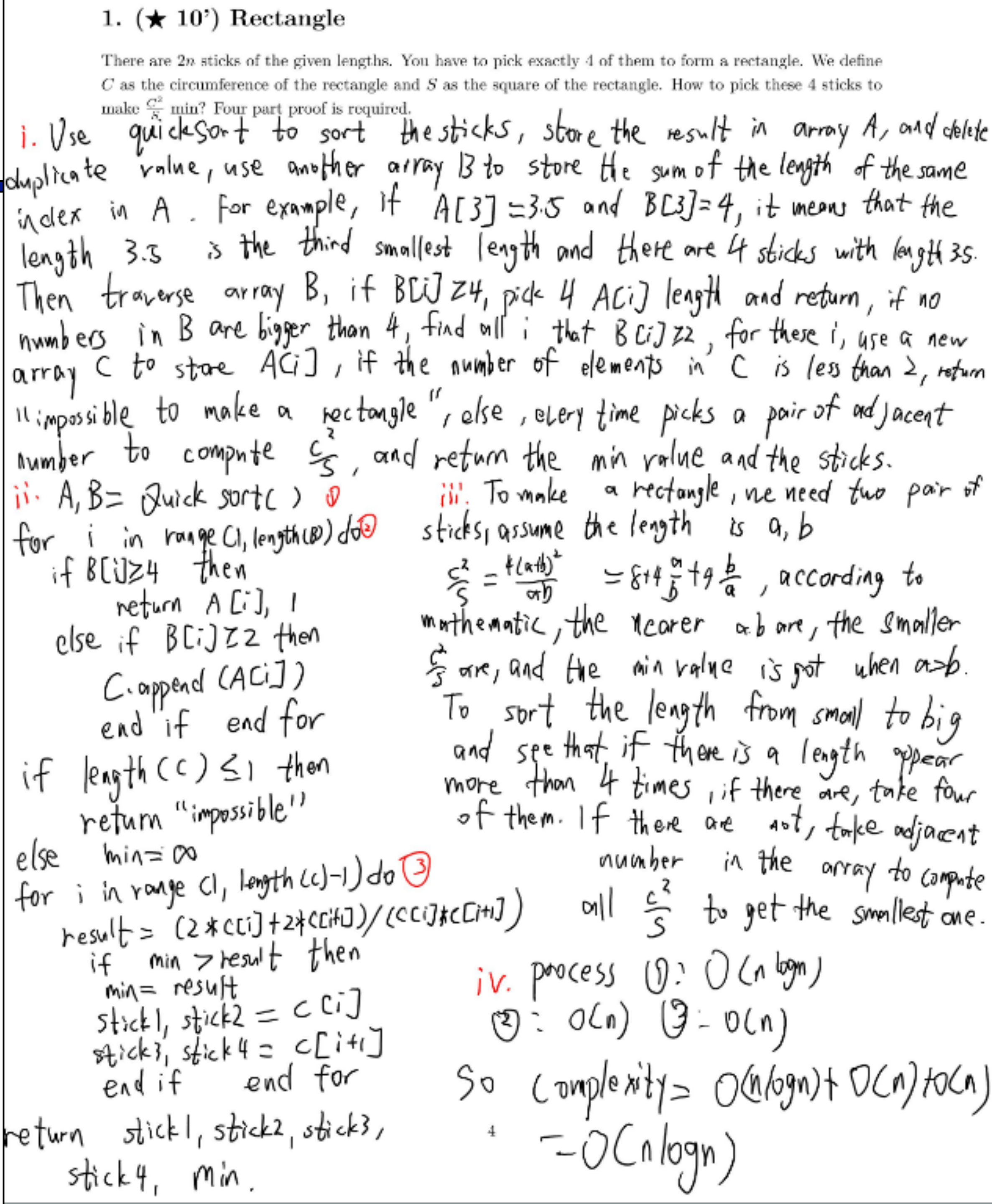
```
Sort sticks as asending order $l_1 \leq l_2 \leq \dots \leq l_n \leq \dots \leq l_{2n}$

for $l$, $i \leq 2n$.
  if $l_i = l_{i+1}$.
  push $l_i$ to leng[ ].
    $j = 1$.
 if length(leng) $\leq 4$
   return "no answer"
     $j \leftarrow j + 5$
while( $j \leq$ length(leng) )
   $a \leftarrow$ leng[j]
   $b \leftarrow$ leng[j+1].
   ans[ ] $\leftarrow \frac{4(a+b)^2}{ab}$
   $j + 2$
```

4

```
return ans[ ].
```

1. Clarity Example 2

- The font is clear, but…
- You should make each part as far as you can.
- You should make your font as tidy as you can.
- You can separate them by lines, which breaks into some subblocks.



1. (★ 10') Rectangle

There are $2n$ sticks of the given lengths. You have to pick exactly 4 of them to form a rectangle. We define C as the circumference of the rectangle and S as the square of the rectangle. How to pick these 4 sticks to make $\frac{C^2}{S}$ min? Four part proof is required.

i. Use quickSort to sort the sticks, store the result in array A, and delete duplicate value, use another array B to store the sum of the length of the same index in A. For example, if $A[3]=3.5$ and $B[3]=4$, it means that the length 3.5 is the third smallest length and there are 4 sticks with length 3.5. Then traverse array B, if $B[i]\geq 4$, pick 4 $A[i]$ length and return, if no numbers in B are bigger than 4, find all i that $B[i]\geq 2$, for these i, use a new array C to store $A[i]$, if the number of elements in C is less than 2, return "impossible to make a rectangle", else, every time picks a pair of adjacent number to compute $\frac{c^2}{S}$, and return the min value and the sticks.

ii.
```
A, B = Quick sort( )  ①
for i in range (1, length(B)) do ②
   if B[i] ≥ 4 then
        return A[i], 1
   else if B[i] ≥ 2 then
        C.append (A[i])
        end if   end for
if length (c) ≤ 1 then
     return "impossible"
else   min = ∞
for i in range (1, length (c)-1) do ③
   result = (2*c[i]+2*c[i+1])/(c[i]*c[i+1])
      if  min > result  then
       min = result
       stick1, stick2 = c[i]
       stick3, stick4 = c[i+1]
       end if      end for
return  stick1, stick2, stick3,
   stick4,  min.
```

iii. To make a rectangle, we need two pair of sticks, assume the length is a, b

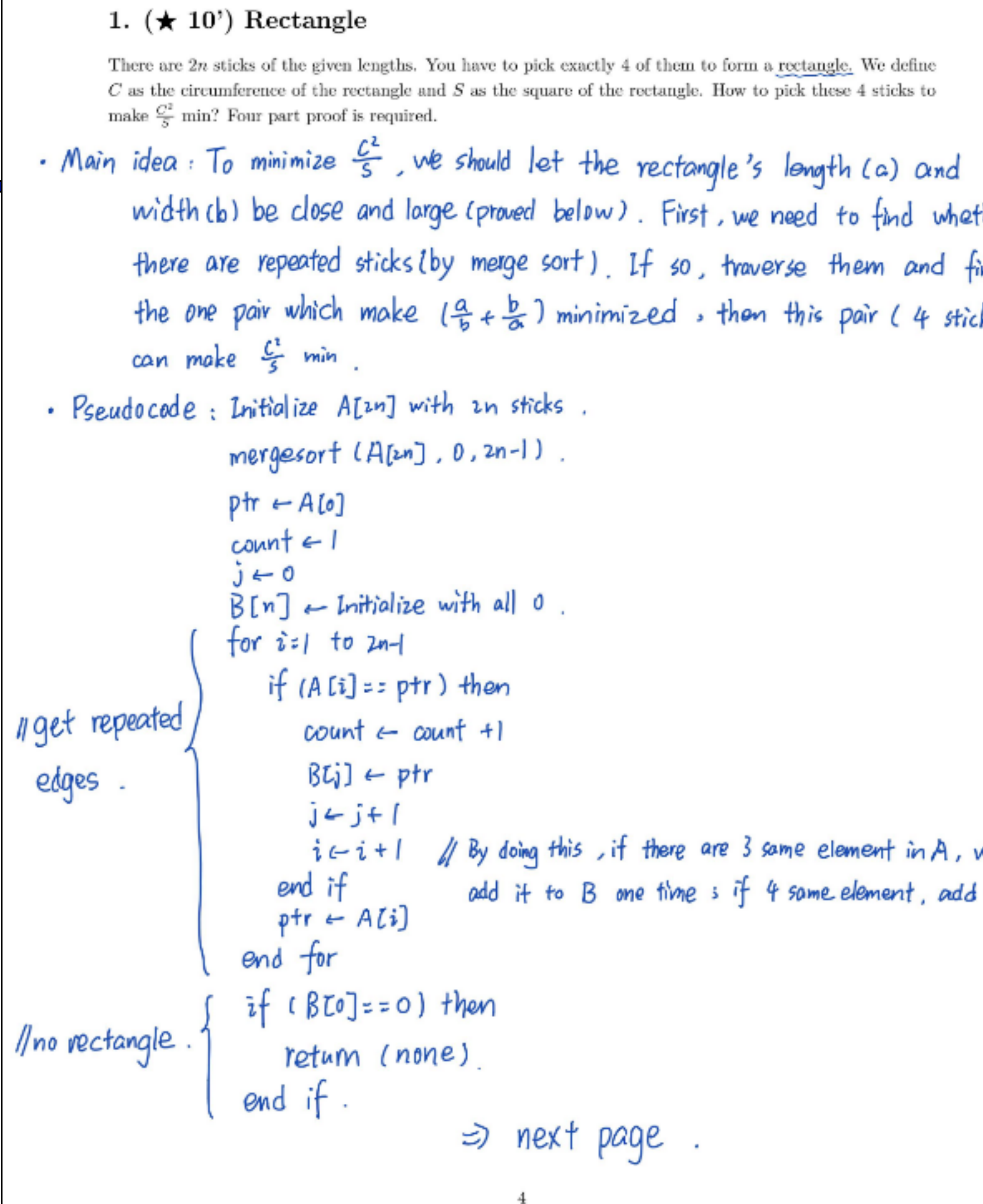
$\frac{c^2}{S}=\frac{4(a+b)^2}{ab}=8+4\frac{a}{b}+4\frac{b}{a}$, according to mathematic, the nearer a.b are, the smaller $\frac{c^2}{S}$ are, and the min value is got when a=b.
To sort the length from small to big and see that if there is a length appear more than 4 times, if there are, take four of them. If there are not, take adjacent number in the array to compute all $\frac{c^2}{S}$ to get the smallest one.

iv. process ①: $O(n\log n)$
②: $O(n)$ ③: $O(n)$
So complexity $= O(n\log n)+O(n)+O(n)$
$= O(n\log n)$

4

1. Clarity Example 3

- This one is much more clear.
- Font size and line spacing are proper.
- Very tight!
- You can easily see where is each part.
- If you can circle/underline the important solution, it's better!



1. (★ 10') Rectangle

There are $2n$ sticks of the given lengths. You have to pick exactly 4 of them to form a rectangle. We define C as the circumference of the rectangle and S as the square of the rectangle. How to pick these 4 sticks to make $\frac{C^2}{S}$ min? Four part proof is required.

- Main idea: To minimize $\frac{C^2}{S}$, we should let the rectangle's length (a) and width (b) be close and large (proved below). First, we need to find whet there are repeated sticks (by merge sort). If so, traverse them and fi the one pair which make $(\frac{a}{b}+\frac{b}{a})$ minimized, then this pair (4 stic can make $\frac{C^2}{S}$ min.

- Pseudocode: Initialize A[2n] with 2n sticks.

```
mergesort (A[2n], 0, 2n-1).
ptr ← A[0]
count ← 1
j ← 0
B[n] ← Initialize with all 0.
for i:=1 to 2n-1
    if (A[i]==ptr) then
        count ← count +1
        B[j] ← ptr
        j ← j+1
        i ← i+1    // By doing this, if there are 3 same element in A, v
    end if            add it to B one time; if 4 same element, add
    ptr ← A[i]
end for
if (B[0]==0) then
    return (none).
end if.
```

//get repeated edges.

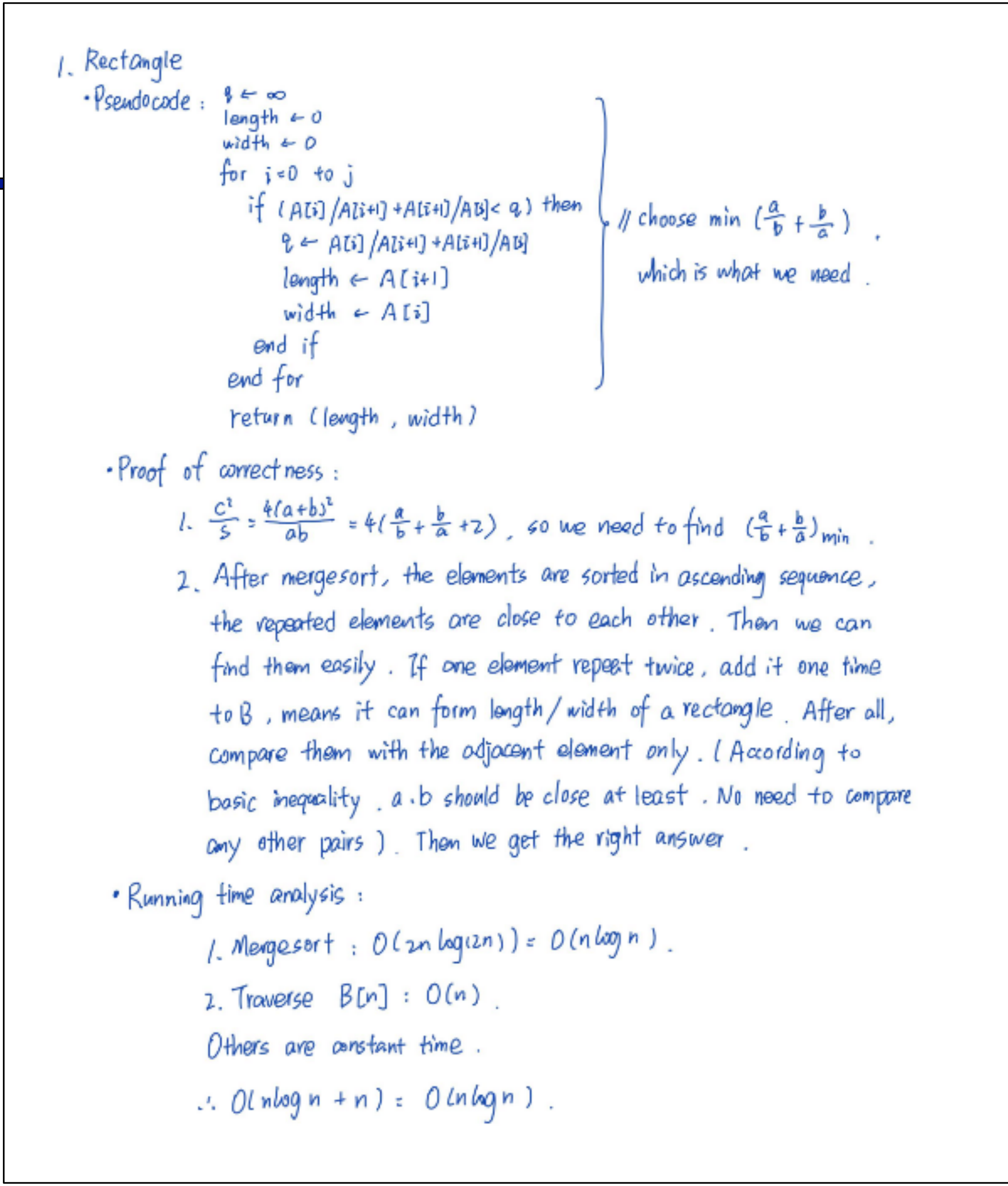
//no rectangle.

⇒ next page.

4

1. Clarity Example 3

- This one is much more clear.
- Font size and line spacing are proper.
- Very tight!
- You can easily see where is each part.
- If you can circle/underline the important solution, it's better!



1. Rectangle

- Pseudocode :

```
q ← ∞
length ← 0
width ← 0
for i=0 to j
   if (A[i]/A[i+1]+A[i+1]/A[i] < q) then       // choose min (a/b + b/a) .
      q ← A[i]/A[i+1]+A[i+1]/A[i]                  which is what we need .
      length ← A[i+1]
      width ← A[i]
   end if
end for
return (length , width)
```

- Proof of correctness :

  1. $\frac{c^2}{S} = \frac{4(a+b)^2}{ab} = 4(\frac{a}{b}+\frac{b}{a}+2)$, so we need to find $(\frac{a}{b}+\frac{b}{a})_{min}$.
  2. After mergesort, the elements are sorted in ascending sequence, the repeated elements are close to each other. Then we can find them easily. If one element repeat twice, add it one time to B, means it can form length / width of a rectangle. After all, compare them with the adjacent element only. (According to basic inequality. a.b should be close at least. No need to compare any other pairs). Then we get the right answer.

- Running time analysis :

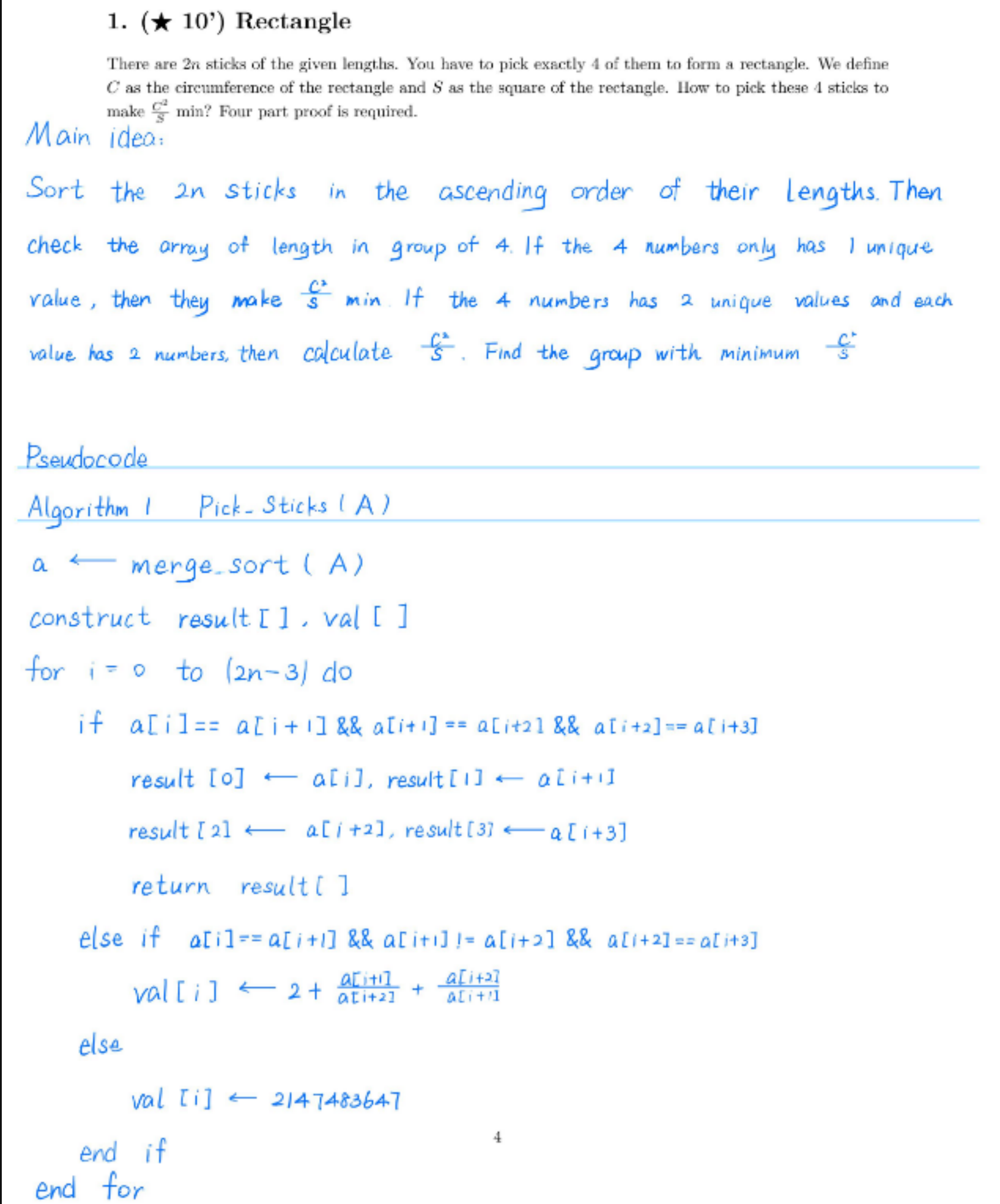
  1. Mergesort : $O(2n\log(2n)) = O(n\log n)$.
  2. Traverse $B[n]$: $O(n)$.

  Others are constant time .

  $\therefore O(n\log n + n) = O(n\log n)$.

1. Clarity Example 4

- This one is much more clear.
- Font size and line spacing are proper.
- Very tight!
- You can easily see where is each part.
- If you can circle/underline the important solution, it's better!



1. (★ 10') Rectangle

There are $2n$ sticks of the given lengths. You have to pick exactly 4 of them to form a rectangle. We define C as the circumference of the rectangle and S as the square of the rectangle. How to pick these 4 sticks to make $\frac{C^2}{S}$ min? Four part proof is required.

Main idea:

Sort the 2n sticks in the ascending order of their lengths. Then check the array of length in group of 4. If the 4 numbers only has 1 unique value, then they make $\frac{C^2}{S}$ min. If the 4 numbers has 2 unique values and each value has 2 numbers, then calculate $\frac{C^2}{S}$. Find the group with minimum $\frac{C^2}{S}$

Pseudocode

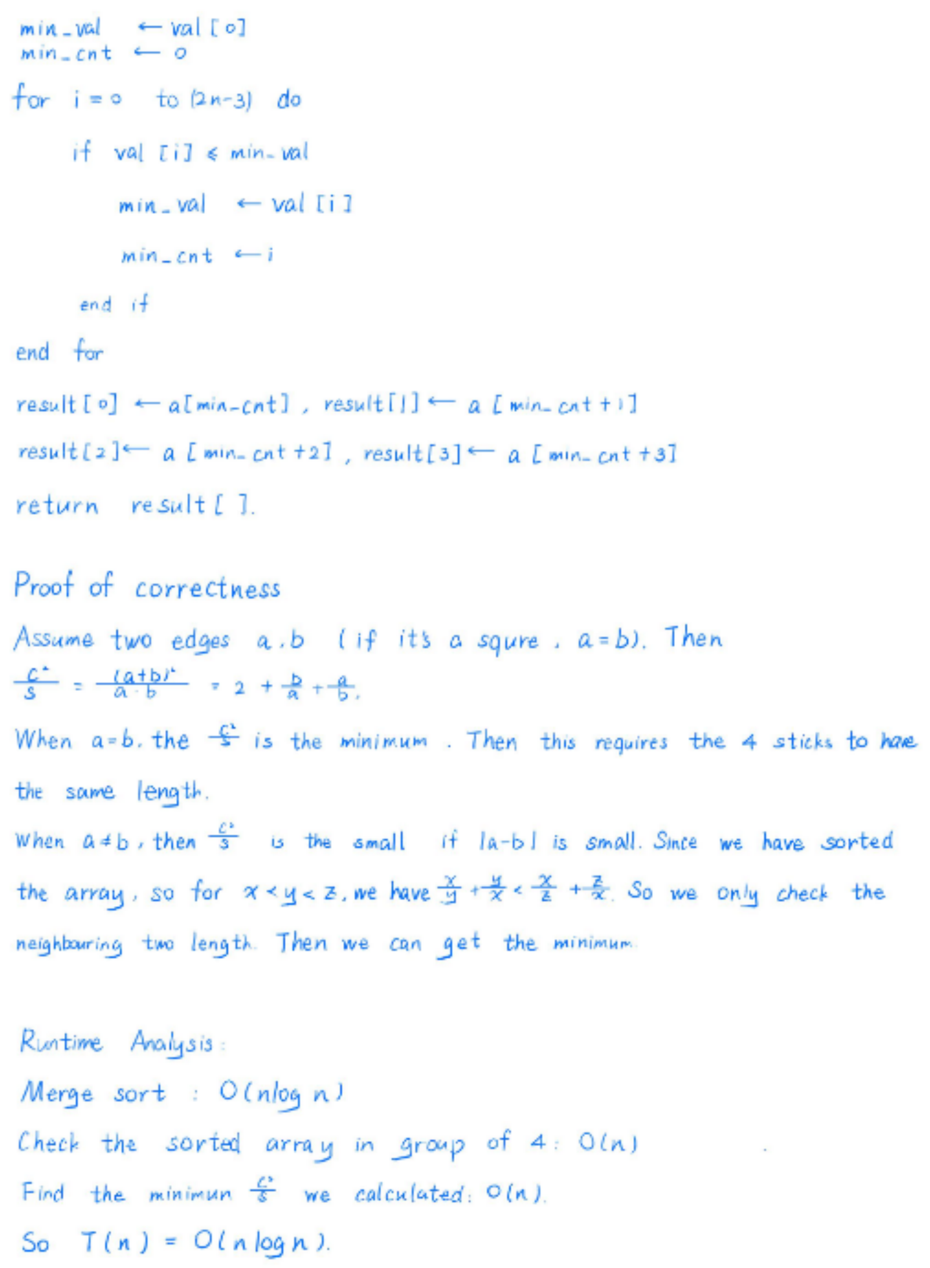
Algorithm 1 Pick_Sticks (A)

```
a ← merge_sort ( A )
construct result [ ] , val [ ]
for i = 0 to (2n-3) do
    if a[i] == a[i+1] && a[i+1] == a[i+2] && a[i+2] == a[i+3]
        result [0] ← a[i], result [1] ← a[i+1]
        result [2] ← a[i+2], result [3] ← a[i+3]
        return result [ ]
    else if a[i] == a[i+1] && a[i+1] != a[i+2] && a[i+2] == a[i+3]
        val [ i ] ← 2 + $\frac{a[i+1]}{a[i+2]}$ + $\frac{a[i+2]}{a[i+1]}$
    else
        val [i] ← 2147483647
    end if
end for
```

4

1. Clarity Example 4

- This one is much more clear.
- Font size and line spacing are proper.
- Very tight!
- You can easily see where is each part.
- If you can circle/underline the important solution, it's better!



```
min_val ← val [0]
min_cnt ← 0
for i = 0 to (2n-3) do
    if val [i] ≤ min_val
        min_val ← val [i]
        min_cnt ← i
    end if
end for
result [0] ← a[min_cnt] , result[1] ← a [min_cnt + 1]
result[2] ← a [min_cnt + 2] , result[3] ← a [min_cnt + 3]
return result [ ].
```

Proof of correctness

Assume two edges a . b (if it's a squre , a = b). Then

$\frac{c^2}{s} = \frac{(a+b)^2}{a \cdot b} = 2 + \frac{b}{a} + \frac{a}{b}.$

When a = b, the $\frac{c^2}{s}$ is the minimum . Then this requires the 4 sticks to have the same length.

When $a \neq b$, then $\frac{c^2}{s}$ is the small if $|a-b|$ is small. Since we have sorted the array, so for $x < y < z$, we have $\frac{x}{y} + \frac{y}{x} < \frac{x}{z} + \frac{z}{x}$. So we only check the neighbouring two length. Then we can get the minimum.

Runtime Analysis:

Merge sort : $O(n \log n)$

Check the sorted array in group of 4 : $O(n)$

Find the minimun $\frac{c^2}{s}$ we calculated: $O(n)$.

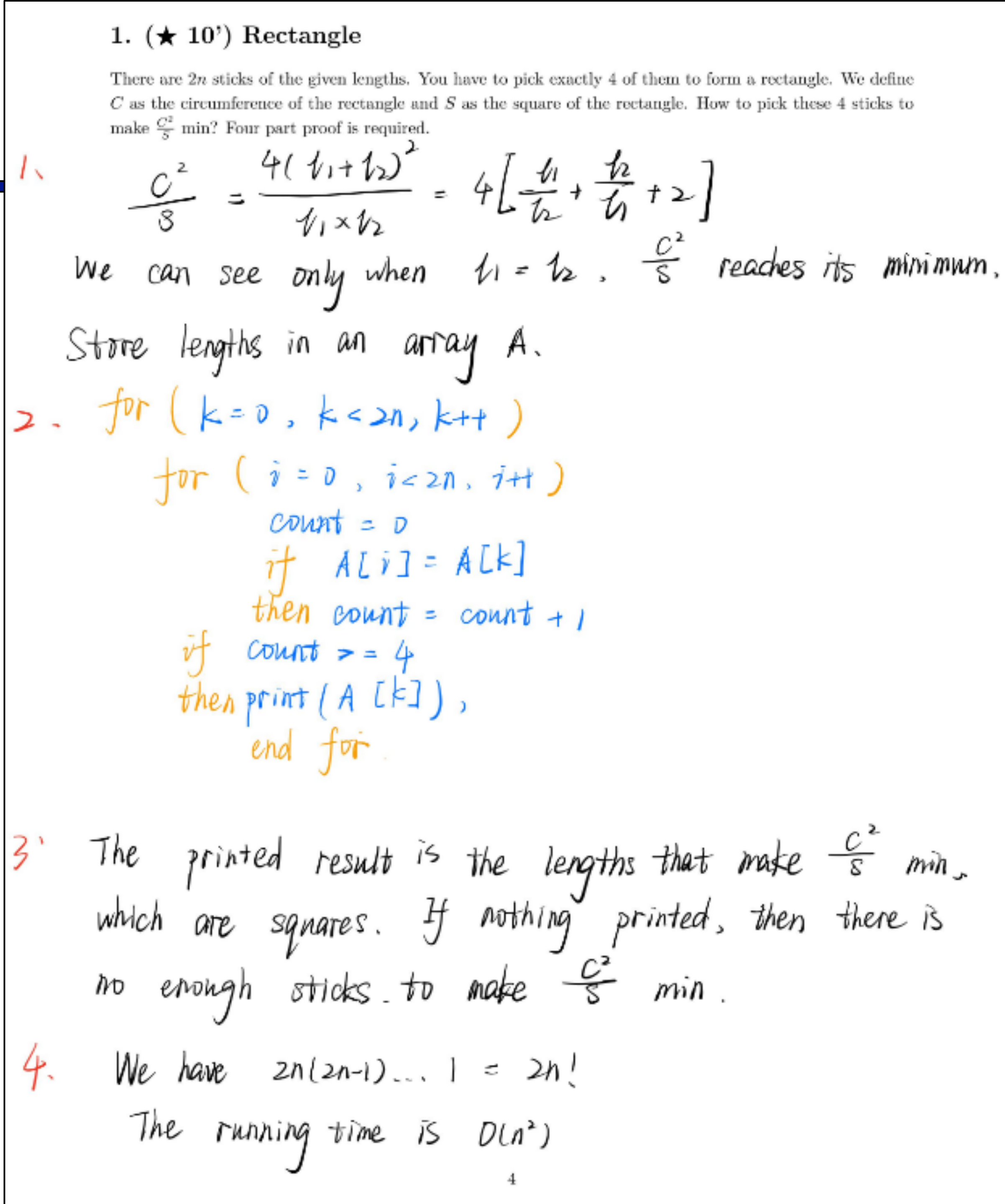
So $T(n) = O(n \log n)$.

2. Write more as you can

- Do not ignore some important steps.
- First, write as more keywords as you can, or something relates to your topic.
  - Divide and Conquer: The recursive function…
  - Greedy: The optimal solution…
  - Dynamic Programming: Subproblem, The Bellman equation…
  - NP-Completeness: The known NPC problem, Reduction, In NP…
- Second, please explain your analysis in details, especially when you analyze the time complexity, prove correctness and show your main idea. Break them into each sub steps which can not be composed.
- But this does not mean the more you write, the score will be higher.
  - Clear process of basic thinking. (重思路清晰，重质！)

2. Detail Example 1

- His main idea is too weak. Using what sorting method? Using what method of algorithm?
- His code has no function title, input parameters, and the value to return.
- His time analysis is too weak. You should analyze how to get time complexity for each step.



1. (★ 10') Rectangle

There are $2n$ sticks of the given lengths. You have to pick exactly 4 of them to form a rectangle. We define C as the circumference of the rectangle and S as the square of the rectangle. How to pick these 4 sticks to make $\frac{C^2}{S}$ min? Four part proof is required.

1、 $\frac{C^2}{S} = \frac{4(l_1+l_2)^2}{l_1\times l_2} = 4[\frac{l_1}{l_2}+\frac{l_2}{l_1}+2]$

We can see only when $l_1 = l_2$, $\frac{C^2}{S}$ reaches its minimum.

Store lengths in an array A.

2.
```
for ( k=0 , k<2n, k++ )
    for ( i=0 , i<2n , i++ )
        count = 0
        if A[i] = A[k]
        then count = count + 1
    if count >= 4
    then print ( A [k] ) ,
        end for
```

3、 The printed result is the lengths that make $\frac{C^2}{S}$ min, which are squares. If nothing printed, then there is no enough sticks to make $\frac{C^2}{S}$ min.

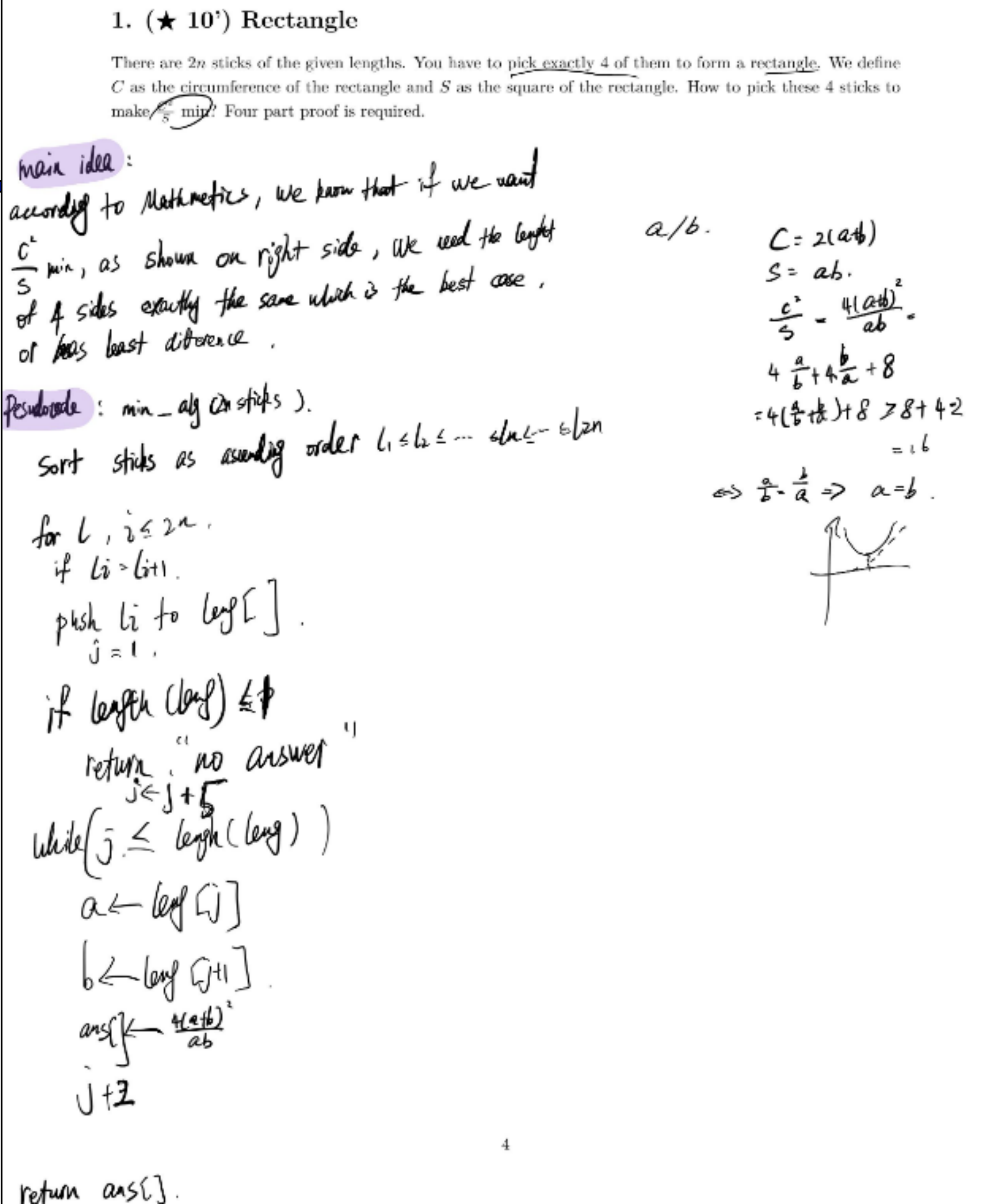
4. We have $2n(2n-1)\dots 1 = 2n!$

The running time is $O(n^2)$

4

2. Detail Example 2

- The main idea is not the key to the solution, i.e., not a/b min.
- He doesn't mention what sorting algorithm he has used in both main idea and code.
- The code looks like a mass.
- The proof is imcomplete.



1. (★ 10') Rectangle

There are $2n$ sticks of the given lengths. You have to pick exactly 4 of them to form a rectangle. We define C as the circumference of the rectangle and S as the square of the rectangle. How to pick these 4 sticks to make $\frac{C^2}{S}$ min? Four part proof is required.

main idea:
according to Mathmetics, we know that if we want $\frac{C^2}{S}$ min, as shown on right side, we need the lenght of 4 sides exactly the same which is the best case, or has least diference.

a/b.

$C = 2(a+b)$
$S = ab$.
$\frac{C^2}{S} = \frac{4(a+b)^2}{ab}$.
$4\frac{a}{b} + 4\frac{b}{a} + 8$
$= 4(\frac{a}{b} + \frac{b}{a}) + 8 \geq 8 + 4 \cdot 2$
$= 16$
$\Leftrightarrow \frac{a}{b} = \frac{b}{a} \Rightarrow a = b$.

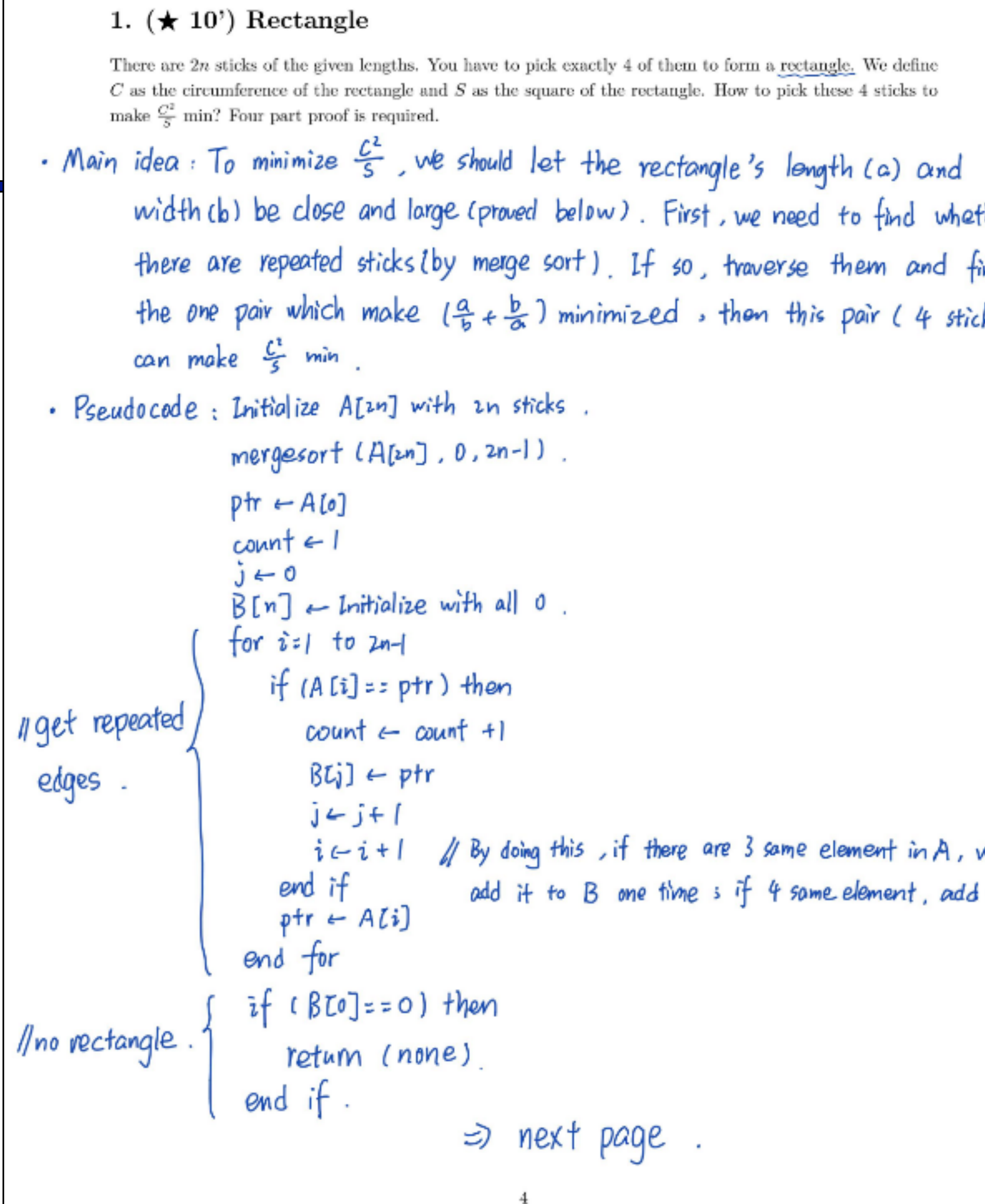
Pesudocode: min_alg (2n sticks).

```
Sort sticks as asending order $l_1 \le l_2 \le \dots \le l_n \le \dots \le l_{2n}$

for l, $i \le 2n$.
  if $l_i = l_{i+1}$.
  push $l_i$ to leng[ ].
    $j = 1$.
  if length(leng) $\le 1$
    return "no answer"
      $j \leftarrow j + 5$
while( $j \le$ length(leng) )
    $a \leftarrow$ leng[j]
    $b \leftarrow$ leng[j+1].
    ans[ ] $\leftarrow \frac{4(a+b)^2}{ab}$
    j+2

return ans[ ].
```

4

2. Detail Example 3

- Good!
- But we still recommend you to write down “Greedy Algorithm”.



1. (★ 10’) Rectangle

There are $2n$ sticks of the given lengths. You have to pick exactly 4 of them to form a rectangle. We define C as the circumference of the rectangle and S as the square of the rectangle. How to pick these 4 sticks to make $\frac{C^2}{S}$ min? Four part proof is required.

- Main idea : To minimize $\frac{C^2}{S}$, we should let the rectangle’s length (a) and width (b) be close and large (proved below) . First , we need to find whet there are repeated sticks (by merge sort) . If so , traverse them and fi the one pair which make $(\frac{a}{b} + \frac{b}{a})$ minimized , then this pair (4 stic can make $\frac{C^2}{S}$ min .

- Pseudocode : Initialize A[2n] with 2n sticks .

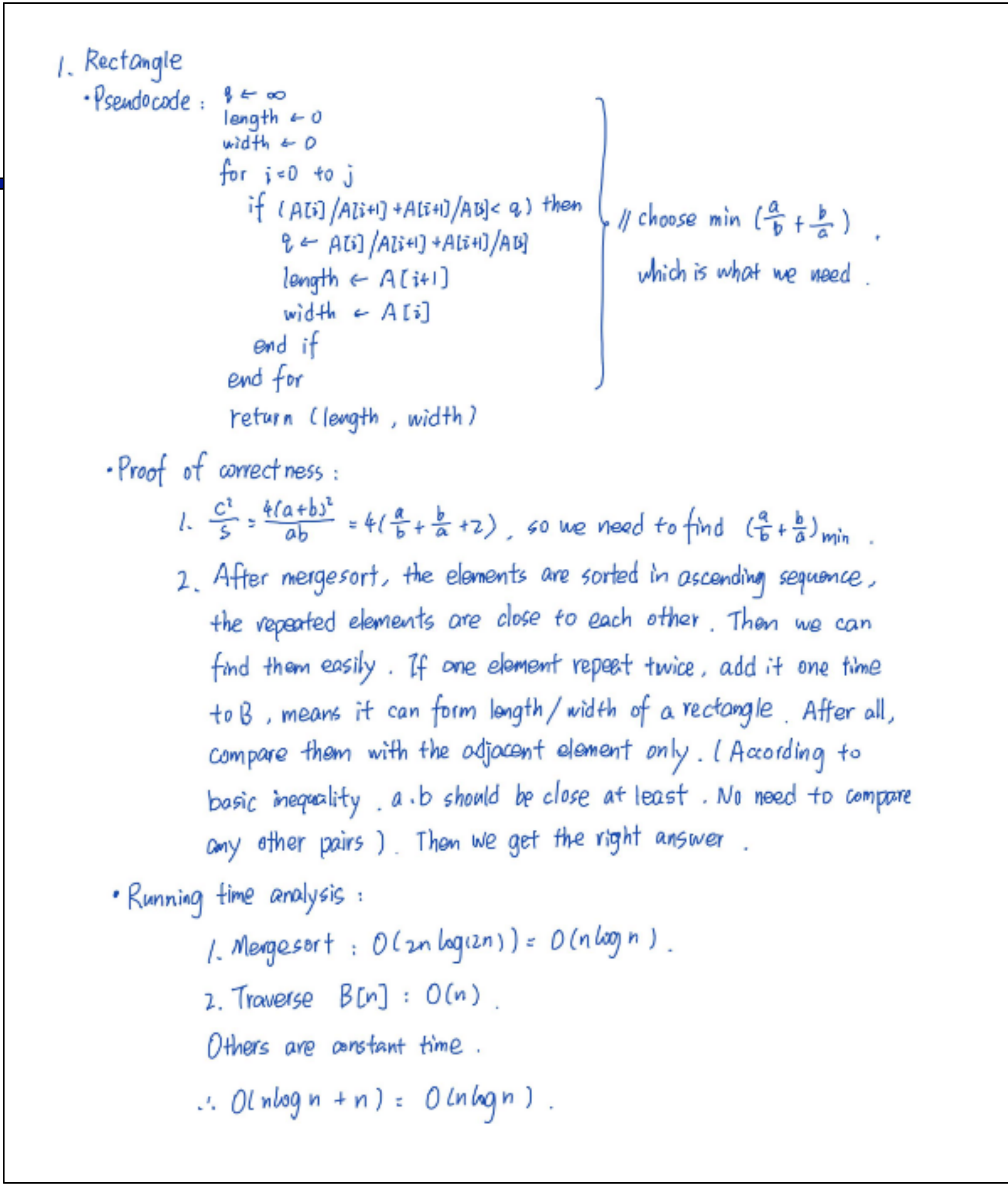
```
                  mergesort (A[2n] , 0 , 2n-1) .
                  ptr ← A[0]
                  count ← 1
                  j ← 0
                  B[n] ← Initialize with all 0 .
                  for i:=1 to 2n-1
//get repeated       if (A[i] == ptr) then
  edges .               count ← count +1
                        B[j] ← ptr
                        j ← j+1
                        i ← i+1   // By doing this , if there are 3 same element in A , v
                     end if          add it to B one time ; if 4 same element , add
                     ptr ← A[i]
                  end for
                  if (B[0]==0) then
//no rectangle .     return (none) .
                  end if .
```

⇒ next page .

4

2. Detail Example 3

- Good!
- But we still recommend you to write down "Greedy Algorithm".



1. Rectangle

- Pseudocode:

```
q ← ∞
length ← 0
width ← 0
for i=0 to j
    if (A[i]/A[i+1]+A[i+1]/A[i]< q) then
        q ← A[i]/A[i+1]+A[i+1]/A[i]
        length ← A[i+1]
        width ← A[i]
    end if
end for
return (length, width)
```

// choose min $(\frac{a}{b}+\frac{b}{a})$. which is what we need.

- Proof of correctness:

1. $\frac{c^2}{s} = \frac{4(a+b)^2}{ab} = 4(\frac{a}{b}+\frac{b}{a}+2)$, so we need to find $(\frac{a}{b}+\frac{b}{a})_{min}$.
2. After mergesort, the elements are sorted in ascending sequence, the repeated elements are close to each other. Then we can find them easily. If one element repeat twice, add it one time to B, means it can form length/width of a rectangle. After all, compare them with the adjacent element only. (According to basic inequality. a.b should be close at least. No need to compare any other pairs). Then we get the right answer.

- Running time analysis:

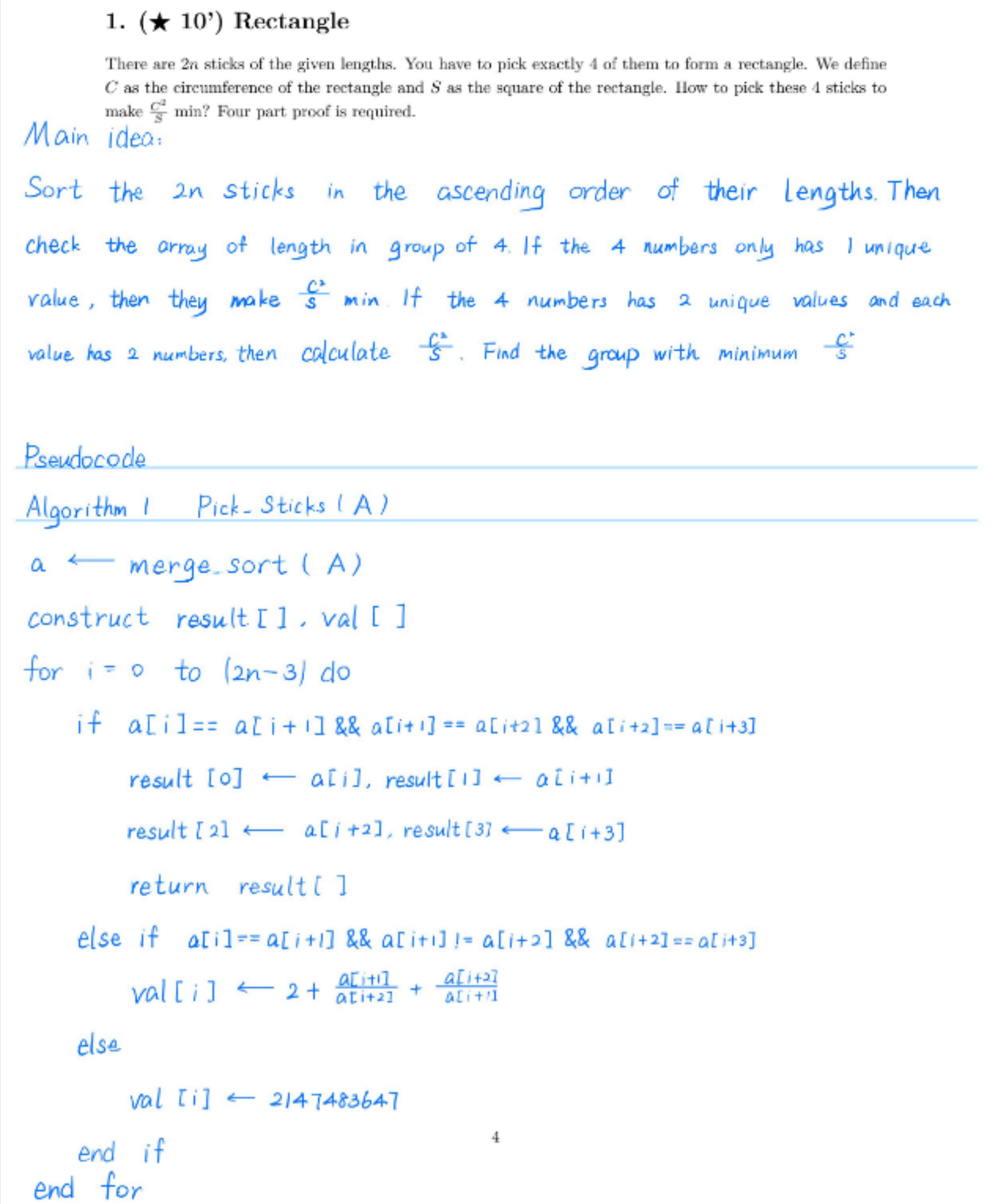
1. Mergesort: $O(2n\log(2n)) = O(n\log n)$.
2. Traverse $B[n]$: $O(n)$.

Others are constant time.

$\therefore O(n\log n + n) = O(n\log n)$.

2. Detail Example 4

- Good!
- But we still recommend you to write down “Greedy Algorithm”.



1. (★ 10’) Rectangle

There are $2n$ sticks of the given lengths. You have to pick exactly 4 of them to form a rectangle. We define C as the circumference of the rectangle and S as the square of the rectangle. How to pick these 4 sticks to make $\frac{C^2}{S}$ min? Four part proof is required.

Main idea:

Sort the 2n sticks in the ascending order of their lengths. Then check the array of length in group of 4. If the 4 numbers only has 1 unique value, then they make $\frac{C^2}{S}$ min. If the 4 numbers has 2 unique values and each value has 2 numbers, then calculate $\frac{C^2}{S}$. Find the group with minimum $\frac{C^2}{S}$

Pseudocode

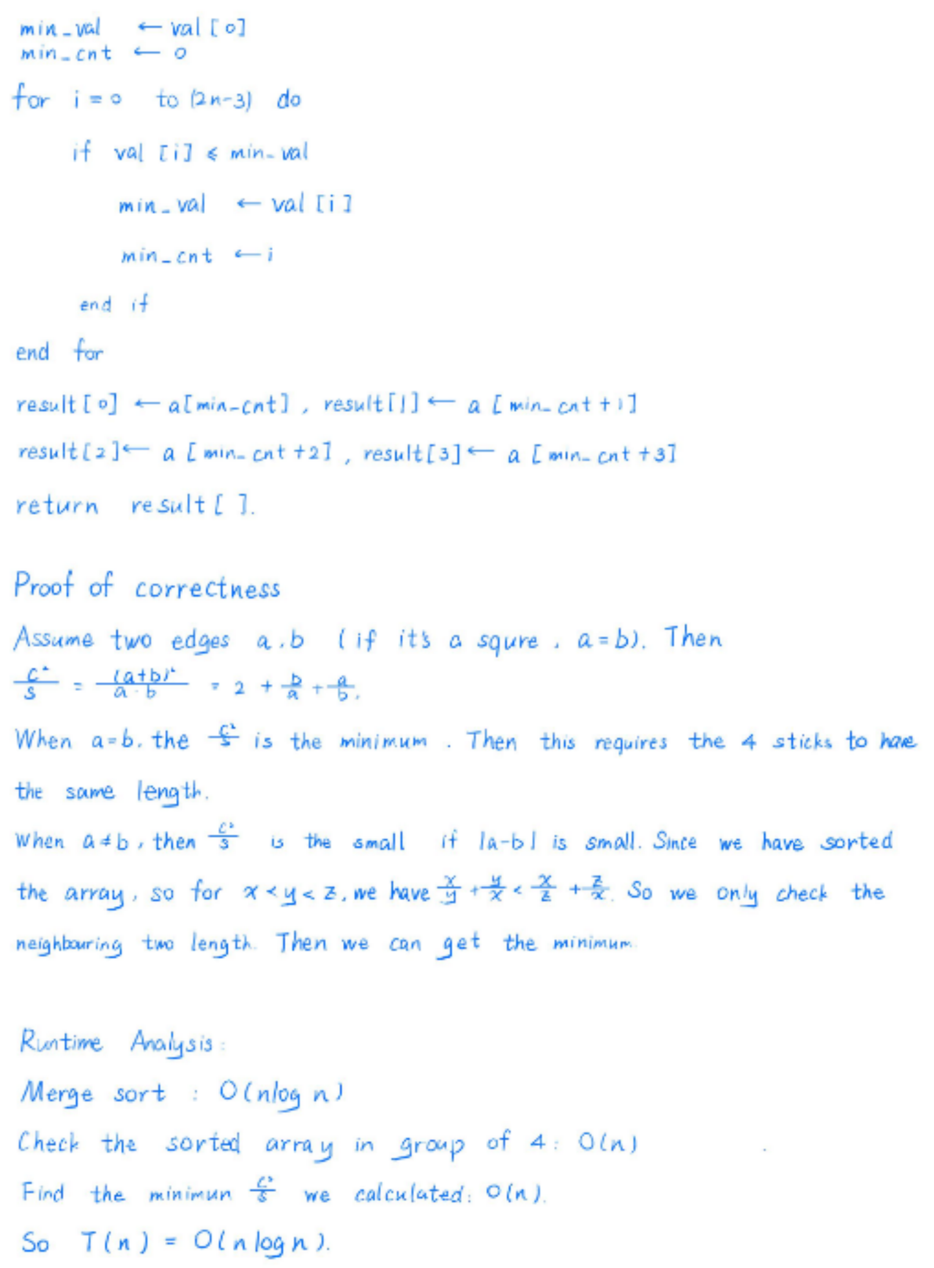
Algorithm 1 Pick_Sticks (A)

```
a ← merge_sort ( A )
construct result [ ] , val [ ]
for i = 0 to (2n−3) do
    if a[i] == a[i+1] && a[i+1] == a[i+2] && a[i+2] == a[i+3]
        result [0] ← a[i], result [1] ← a[i+1]
        result [2] ← a[i+2], result [3] ← a[i+3]
        return result [ ]
    else if a[i] == a[i+1] && a[i+1] != a[i+2] && a[i+2] == a[i+3]
        val [i] ← 2 + $\frac{a[i+1]}{a[i+2]}$ + $\frac{a[i+2]}{a[i+1]}$
    else
        val [i] ← 2147483647
    end if
end for
```

4

2. Detail Example 4

- Good!
- But we still recommend you to write down “Greedy Algorithm”.



```
min_val ← val[0]
min_cnt ← 0
for i = 0 to (2n-3) do
    if val[i] ≤ min_val
        min_val ← val[i]
        min_cnt ← i
    end if
end for
result[0] ← a[min_cnt], result[1] ← a[min_cnt+1]
result[2] ← a[min_cnt+2], result[3] ← a[min_cnt+3]
return result[ ].
```

Proof of correctness

Assume two edges a, b (if it's a squre, a = b). Then

$\frac{C^2}{S} = \frac{(a+b)^2}{a \cdot b} = 2 + \frac{b}{a} + \frac{a}{b}$.

When $a = b$, the $\frac{C^2}{S}$ is the minimum. Then this requires the 4 sticks to have the same length.

When $a \neq b$, then $\frac{C^2}{S}$ is the small if $|a-b|$ is small. Since we have sorted the array, so for $x < y < z$, we have $\frac{x}{y} + \frac{y}{x} < \frac{x}{z} + \frac{z}{x}$. So we only check the neighbouring two length. Then we can get the minimum.

Runtime Analysis:

Merge sort : $O(n\log n)$

Check the sorted array in group of 4: $O(n)$

Find the minimun $\frac{C^2}{S}$ we calculated: $O(n)$.

So $T(n) = O(n \log n)$.

Thanks for your hard work!

Wish you have a better score in final exam!

CS101 Teaching Team