Algorithms Middle Examination

Nov. 1, 2024 (10:10 ~ 12:20)

(If you design an algorithm, you must have a pseudo code to show your algorithm and analyze the algorithm's time complexity in the worst case. It would help to put comments after your pseudo code to clarify your algorithm.)

1. (Ch. 2) (10%) Use mathematical induction to show that when $n \geq 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & if\ n = 2, \\ 2T\left(\frac{n}{2}\right) + n & if\ n > 2 \end{cases}$$

is $T(n) = n \lg n$.

Answer:

Base case:

When n = 2, $T(2) = 2lg2 = 2$. The solution holds.

Induction:

Assume $T(n) = nlgn$, then there exists k > 2 such that $T(2^k) = 2^k lg2^k$

We then prove n = 2^{k+1} hold when k > 2

$$\begin{aligned} T(2^{k+1}) &= 2T(2^k) + 2^{k+1} \\ &= 2(2^k\ lg2^k) + 2^{k+1} \\ &= 2^{k+1}(lg2^k + 1) \\ &= 2^{k+1}(lg2^{k+1}) \end{aligned}$$

Since $T(n) = nlgn$ hold when n = 2^{k+1}, we prove that our assumption is correct.

$T(n) = nlgn$ holds when $n \geq 2$

配分：

寫出 base case 證明給 2 分

有假設當 k > 2 時$T(n) = nlgn$成立給 2 分

剩餘證明 6 分

2. (Ch. 3) (6%) Explain why the statement, “The running time of algorithm *A* is at least *O*(*n*),” is meaningless.

Answer:

Since Big-O notation describes an upper bound, not a lower bound. To express a lower bound, Big-Omega(Ω) notation should be used.

3. (Ch. 4) (6%) Your friend has tried to prove 1+2+…+ $n = O(n)$. His proof is by induction as follows. Basis: $1 = O(n)$, for $n = 1$.
   Assume $n = k$ is hold. That is $1 + 2 + \ldots + k = O(n)$.
   Induction ($n = k + 1$): $1 + 2+ \ldots+ k + (k+1) = O(n) + (k+1) = O(n) + O(n) = O(2n) = O(n)$
   So, $1 + 2+ \ldots + n = O(n)$. Please indicate the bug of your friend's proof.

   Answer:

   When $n = k$ is hold, $1 + 2 + \cdots + k = O(n) = O(k) \leq ck$.

   So, when $n = k + 1$, $1 + 2 + \cdots + k + (k + 1) \leq c(k + 1) = ck + c$.

   But in this induction, $1 + 2 + \cdots + k + (k + 1) = O(n) + (k + 1)$

   $$\leq ck + (k + 1)$$

   $$\neq ck + c$$

   Therefore, the induction is wrong.

4. (Ch. 4) (6%) Solve the following recurrence by changing variables: $T(n) = 2T(\sqrt{n}) + \theta(1)$.

   Answer:

   Set $m = lgn$, we get

   $$T(2^m) = 2T(2^{m/2}) + \theta(1)$$

   Next, rename $S(m) = T(2^m) = T(n)$

   $$S(m) = 2S(m/2) + \theta(1)$$

   Then, we solve $S(m)$.

   By recursion tree method

   Assume $\theta(1) = d$ we have:

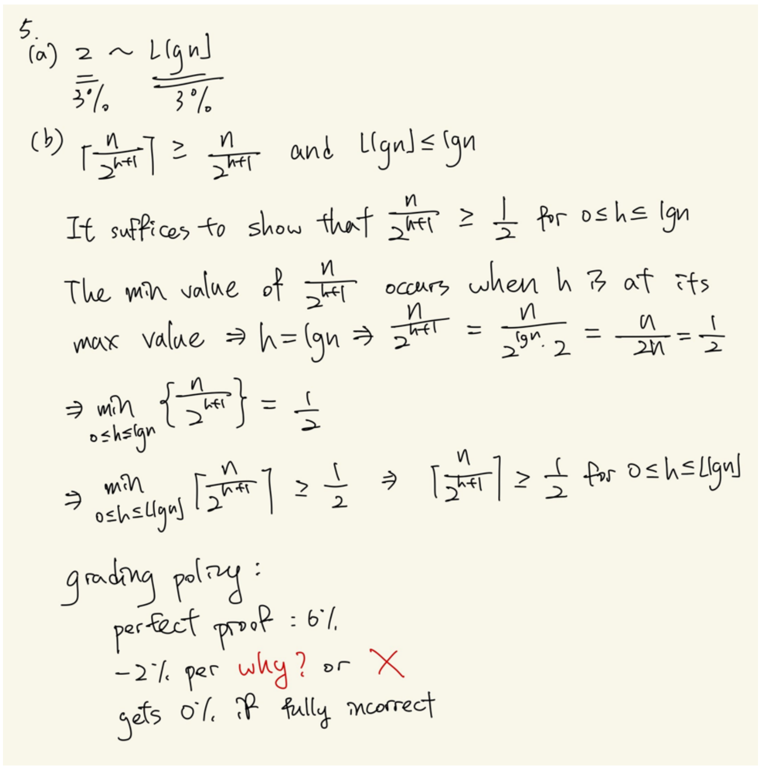
   $$S(m) = 2S(m/2) + d$$

$$= 2(2S(m/4) + d) + d$$
$$= 4S(m/4) + 3d$$
$$= 4(2S(m/8) + d) + 3d$$
$$= 8\big(S(m/8)\big) + 7d$$
$$= \cdots$$
$$= m \cdot S(1) + (m-1)d$$
$$= m \cdot d + (m-1)d$$
$$= 2md - d$$
$$= O(m)$$

By $m = lgn$, $\mathrm{T(n)} = S(m) = O(m) = O(lgn)$.

5. (Ch. 6) (6%) a) At which levels in a min-heap might the kth largest element reside, for $2 \le k \le \lfloor n/2 \rfloor$, assuming that all elements are distinct. (6%) b) Show that $\lceil n/2^{h+1} \rceil \ge 1/2$ for $0 \le h \le \lfloor \lg n \rfloor$.

Answer:



5.

(a) $\underset{3\%}{\underline{\underline{2}}} \sim \underset{3\%}{\underline{\underline{\lfloor lg n \rfloor}}}$

(b) $\left\lceil \frac{n}{2^{h+1}} \right\rceil \ge \frac{n}{2^{h+1}}$ and $\lfloor lg n \rfloor \le lg n$

It suffices to show that $\frac{n}{2^{h+1}} \ge \frac{1}{2}$ for $0 \le h \le lgn$

The min value of $\frac{n}{2^{h+1}}$ occurs when h is at its max value $\Rightarrow h = lgn \Rightarrow \frac{n}{2^{h+1}} = \frac{n}{2^{lgn} \cdot 2} = \frac{n}{2n} = \frac{1}{2}$

$\Rightarrow \min\limits_{0 \le h \le lgn} \left\{ \frac{n}{2^{h+1}} \right\} = \frac{1}{2}$

$\Rightarrow \min\limits_{0 \le h \le \lfloor lgn \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \ge \frac{1}{2} \Rightarrow \left\lceil \frac{n}{2^{h+1}} \right\rceil \ge \frac{1}{2}$ for $0 \le h \le \lfloor lgn \rfloor$

grading policy:

perfect proof : 6%

−2% per why? or ✗

gets 0% if fully incorrect

6. (Ch. 7) (12%) Please prove that the average-case time complexity of Quicksort is $O(n \lg n)$.

Answer:

6. $X =$ # comparisons

average-case time complexity $= O(n + E[X])$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \underline{E[X_{ij}]} \rightarrow \text{comparisons between } a_i, a_j$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=2}^{n-i+1} \frac{2}{j}$$

$$\le \sum_{i=1}^{n-1} \lg n$$

$$\le n \lg n$$

$\Rightarrow$ average-case time complexity $= O(n + n\lg n) = O(n \lg n)$

grading policy:

perfect proof : 12%

−3% per why? or X

gets 0% if fully incorrect

7. (Ch. 8) (12%) It is known that $\Omega(n \log n)$ is a lower bound for sorting problems. However, we have seen algorithms like counting sort or radix sort, which can sort n items in $O(n)$ time. Is there a contradiction? If not, please explain. Can we conclude that the performance of counting sort or radix sort is better than merge sort or heapsort?

Answer:

7. There is no contradiction, $\Omega(n\lg n)$ is the lower bound of comparison sorts, and counting sort and radix sort are not comparison sort.

No, because the time complexity of counting sort and radix sort also depends on the range of inputs, which can make its performance worse than merge sort or heapsort. (for counting sort, range $(k) > n\lg n$, the complexity will be $\Omega(n\lg n)$. for radix sort, number of digits $(d) > \lg n$, the complexity will be $\Omega(n\lg n)$.)

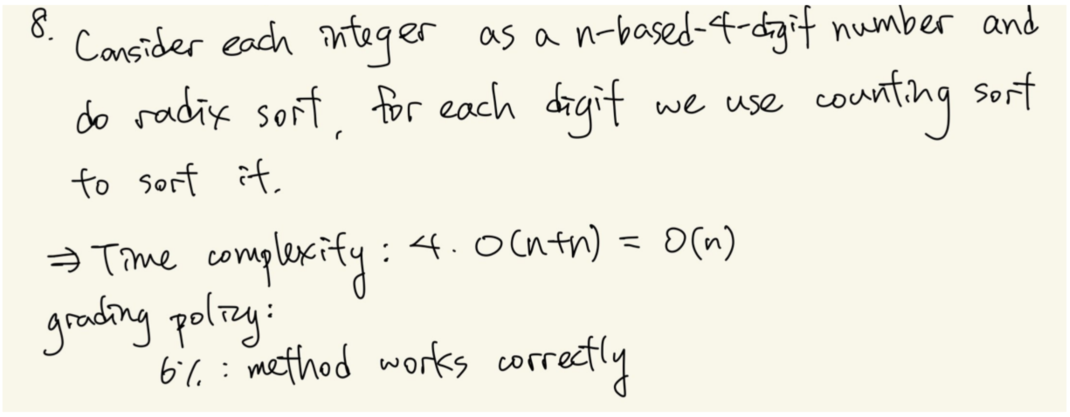
grading policy:

first question : 6%, 3% for no, 3% for explanation

second question: 6%, 3% for no, 3% for explanation

8. (Ch. 8) (6%) Show how to sort *n* integers in the range 0 to $n^4 - 1$ in $O(n)$ time.

Answer:



8. Consider each integer as a n-based-4-digit number and do radix sort, for each digit we use counting sort to sort it.

$\Rightarrow$ Time complexity : $4 \cdot O(n+n) = O(n)$

grading policy:

6% : method works correctly

9. (Ch. 9) (12%) Describe an $O(n)$-time algorithm that, given a set *S* of *n* distinct numbers and a positive integer $k \le n$ determines the *k* numbers in *S* that are closest to the median of *S*.

Answer:

```
Algorithm FindKClosestToMedian(S, k):
    Input: S → a set of n distinct numbers
           k → a positive integer, k <= n
    Output: The k numbers in S that are closest to the median of S
    1. median = MedianOfMedians(S):
    2. S = $|S_i - median|$
    3. Closest_k = SELECT(S)
End Algorithm
```

Step1: Find median from set S → O(n)

Step2: For all number x_i in set S minus median and get absolute. ($x_i = |x_i - median|$) → O(n)

Step3: Use SELECT algorithm to find the kth smallest number. Then, all the left numbers are the closest numbers to the median of S. → O(n)

配分：

方法能符合題目要求但時間複雜度無法在 average case O(n)解出來，給 6 分

方法能符合題目要求且時間複雜度可以在 average case O(n)解出來，給 12 分

方法解釋不夠仔細但大致正確，給 8 分

10. (Ch. 14) (12%) Modified Rod-Cutting Problem: Given a rod of length n inches and a table of prices p_i for $i = 1,2,...,n$, each cut incurs a fixed cost of c. The revenue is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic programming algorithm to output both the optimal cutting approach and maximum revenue r_n. First, you need to provide the problem formulation in recursive form. Then, a pseudocode is given to solve the problem formulation. Finally, analyze your algorithm's time complexity. Please use the following variables with your pseudocode.

Variable	Definition
c	Fixed cost incurred for each cut
p_i	Price of a rod of length i
r_j	Maximum revenue obtainable for a rod of length j
s_j	Optimal cutting position for a rod of length j

Answer:

(3%) Recurrence relation: For each rod length j,

$$r[j] = max(p[j], \max_{1 \le i \le j-1}(p[i] + r[j - i] - c)).$$

(-1%) If you didn't handle the case that didn't cut.

(6%) Pseudo code

```
1.  Procedure MODIFIED-CUT-ROD($p, n, c$)
2.  Let $r[0:n]$ and $s[1:n]$ be a new array
3.  $r[0] = 0$
4.  $for\ j = 1$ to $n$:   // for increasing rod length $j$
        (-1%) If you didn't handle the case that didn't cut.
5.      $q = p[j]$   // handle the case that didn't cut
6.      $s[j] = j$
7.      $for\ i = 1$ to $j - 1$:   // $i$ is the position of the first cut
8.          $if\ q < p[i] + r[j - i] - c$:
9.              $q = p[i] + r[j - i] - c$
10.             $s[j] = i$   // best cut location so far for length $j$
11.     $r[j] = q$   // remember the solution value for length $j$
12. print($r[n]$)   // output maximum revenue
    (-2%) If you didn't output the optimal cutting approach.
13. $while\ n > 0$:
```

```
14.        print($s[n]$)   // cut location for length  n
15.        $n = n - s[n]$   // length of the remainder of the rod
16. End Procedure
```

(3%) For each rod length j, we check all possible cut positions from 1 to $j - 1$, and we need to calculate from 1 to n. Thus, the nested loops results in an overall time complexity of $\Theta(n^2)$.

11. (Chapter 15) (12%) Determine the cost and structure of an optimal binary search tree for four keys ($k1 < k2 < k3 < k4$) with the following probabilities: (Please show your computational steps in detail. You need to show the three tables: $e[i, j]$, $w[i, j]$, and $root[i, j]$.)

i	0	1	2	3	4
pi		0.15	0.10	0.15	0.25
qi	0.05	0.10	0.05	0.05	0.10

Answer:

(3% for each table)

$e[i, j]$

0	1	2	3	4	j / i
0.05	0.45	0.9	1.45	2.45	1
	0.1	0.4	0.9	1.75	2
		0.05	0.35	1.05	3
			0.05	0.55	4
				0.1	5

$w[i, j]$

0	1	2	3	4	j / i
0.05	0.3	0.45	0.65	1	1
	0.1	0.25	0.45	0.8	2
		0.05	0.25	0.6	3
			0.05	0.4	4
				0.1	5

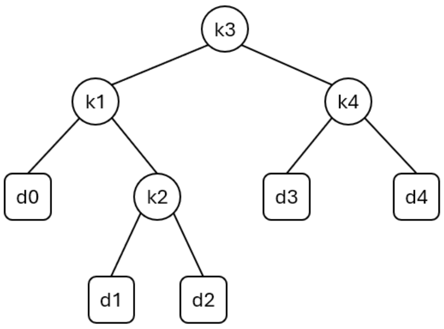
$root\ [i, j]$

1	2	3	4	j / i
1	1	2	3	1
	2	2	3	2
		3	4	3
			4	4

(3%)

The cost of an optimal binary search tree is 2.45.

The structure of an optimal binary search tree is as below


k3
k1
k4
d0
k2
d3
d4
d1
d2