Midterm Solutions

Contact TAs: `ada01@csie.ntu.edu.tw`

Problem 1: Short Answer Questions (20 points)

(a) (3 points) True or False: To prove the correctness of a greedy algorithm, we must prove that every optimal solution contains our greedy choice.

Answer: False. We need to prove that our solution (with greedy choice) is as good as other optimal solutions that may or may not contain the greedy choice.

(b) (3 points) True or False: Any dynamic programming algorithm that solves n subproblems must run in $\Omega(n)$ time.

Answer: True. It must take $\Omega(1)$ time to solve a subproblem, so the total time is $\Omega(n)$.

(c) (4 points) Please explain why the 0/1 knapsack problem may not be solved in $O(nW)$ time (where n is the number of items and W is an integer representing the knapsack's capacity) when items have non-integer weights.

Answer: When items have non-integer weights, there may not be overlapping subproblems (the number of possible sum of weights can be as large as 2^n).

If it can be solved in $O(nW)$ time, we can divide the knapsack's capacity and all the weights of items by W, obtaining an equivalent new problem which can be solved in $O(n \cdot 1) = O(n)$ time. That will imply $\mathcal{P} = \mathcal{NP}$, as 0/1 knapsack is $\mathcal{NP}$-Complete.

(d) (5 points) Recall that in the interval scheduling problem, we want to find the maximum number of compatible jobs given n job requests and their start times $s[i]$ and finish times $f[i]$ for all $1 \le i \le n$. Please provide a counterexample showing that a shortest-interval-first greedy strategy does not always lead to an optimal solution.

Answer:

i	$s[i]$	$f[i]$	length
1	1	5	4
2	3	6	3
3	5	10	5

Optimal solution : Select job 1 and 3 (2 jobs).

Shortest-interval-first : Can only select job 2 (1 job), which is suboptimal.

(e) (5 points) You're working on a dynamic programming problem that has a recurrence relation $A(i, j) = F(A(\lfloor i/2 \rfloor, j), A(i, \lfloor j/2 \rfloor))$, where F is a known function that can be evaluated in constant time, and $A(i, j) = 0$ when $i = 0$ or $j = 0$. To compute $A(m, n)$ for some m and n, you can use either a top-down or a bottom-up method. Which one is more efficient in solving this problem?

Answer: Top-down method is more efficient.

Top-down method only calculates the needed subproblems, which are of the form $A(\lfloor \frac{m}{2^i} \rfloor, \lfloor \frac{n}{2^j} \rfloor)$. There are only $\Theta(\log m \log n)$ such subproblems, hence its time complexity is only $\Theta(\log m \log n)$.

Buttom-up method will require computing every subproblem, resulting in time complexity $\Theta(mn)$.

(In this problem, the indices of needed subproblems can be easily predicted, so one can achieve $\Theta(\log m \log n)$ if you only calculate these needed subproblem in bottom-up method. If your answer is "The same" because of this reason, you will get full credit.)

Problem 2: Asymptotic Notions (10 points)

(a) (5 points) Three divide-and-conquer algorithms are proposed to solve the same problem. Suppose they are all correct, what are their running time (in big-O notation) and which one is more efficient? Please justify your answer.

- Algorithm A divides the problem of size n into three subproblems of size $n/2$ and combines the solutions in linear time.
- Algorithm B divides the problem of size n into two subproblems of size $n-1$ and combines the solutions in constant time.
- Algorithm C divides the problem of size n into nine subproblems of size $n/3$ and combines the solutions in $O(n^2)$ time.

Answer:

- For algorithm A, $T(n) = 3T(n/2) + n$. So
$$T(n) = n + 3/2n + (3/2)^2 n + \cdots = \sum_{k=0}^{\lg n} (3/2)^k n$$
And
$$O\left(\sum_{k=0}^{\lg n} (3/2)^k\right) = O\left(\frac{(3/2)^{\lg n} - 1}{1/2}\right) = O(3^{\lg n}/2^{\lg n}) = O(n^{\lg 3}/n)$$
Hence $T(n) = O(n^{\lg 3}/n)O(n) = O(n^{\lg 3})$.
- For algorithm B, $T(n) = 2T(n-1) + 1$. Then
$$T(n) + 1 = 2(T(n-1)+1) = 4(T(n-2)+1) = \cdots = O(2^n)(T(1)+1) = O(2^n)$$
So $T(n) = O(2^n)$.
- For algorithm C, $T(n) = 9T(n/3) + n^2$. Then
$$T(n) = n^2 + n^2 + n^2 + \cdots = (\log_3 n)n^2 = O(n^2 \lg n)$$

(b) (5 points) Does $f(n) = O(g(n))$ imply $2^{f(n)} = O(2^{g(n)})$? Please justify your answer.

Answer: No, for example, let $f(n) = 2n, g(n) = n$. Since $f(n) = 2g(n)$ so $f(n) = O(g(n))$. But $2^{f(n)} \neq O(2^{g(n)})$, since if exists c, N s.t. $n > N \implies c2^n \geq 2^{2n}$. Then c shall be greater than 2^n for all $n > N$. But $2^n \to \infty$ as $n \to \infty$ which leads to a contradiction, so $2^{f(n)} \notin O(2^{g(n)})$.

Problem 3: Integer Multiplication (10 points)

Given two n-bit integers x and y, please write pseudocode to solve integer multiplication by dividing the original n-bit problem (xy) into three $\frac{n}{2}$-bit subproblems, recursively. Your algorithm should run in $O(n^{\lg 3})$ time.

Answer: See the solution of HW1. Your solution must

- {Discribe, Give a pseudocode} of a correct algorithm. [7 points]
- Justify the running time is $T(n) = 2T(n) + O(n) \implies T(n) = O(n \log n)$. [3 points]

Problem 4: Christmas Again (20 points)

Christmas is approaching. You're helping Santa Clauses to distribute gifts to children.

For ease of delivery, you are asked to divide n gifts into two groups such that the weight difference of these two groups is minimized. The weight of each gift is a positive integer.

(a) (10 points) Please design an algorithm to find an optimal division minimizing the value difference. The algorithm should find the minimal weight difference as well as the groupings in $O(nS)$ time, where S is the total weight of these n gifts.

Hint: This problem can be converted into making one set as close to $S/2$ as possible.

Answer: We consider an equivelant problem of making one set as close to $W = \lfloor S/2 \rfloor$ as possible.

Define $FD(i, w)$ to be the minimal gap between the weight of the bag and W when using the first i gifts only. WLOG, we can assume the weight of the bag is always less than or equal to W.

Then fill the DP table for $0 \leq i \leq n$ and $0 \leq w \leq W$ in which $F(0, w) = w$ for all w, and

$$\begin{aligned} FD(i,w) &= \min\{FD(i-1, w-w_i) - w_i, FD(i-1, w)\} \text{ if } FD(i-1, w-w_i) \geq w_i \\ &= FD(i-1, w) \text{ otherwise} \end{aligned}$$

This takes $O(nS)$ time. $FD(n, W)$ is the minimum gap.

Finally, to reconstruct the answer, we backtrack from (n, W). During backtracking, if $FD(i, j) = FD(i-1, j)$ then i is not selected in the bag and we move to $F(i-1, j)$. Otherwise, i is selected and we move to $F(i-1, j-w_i)$.

We show that this problem has optimal substructure.
Proof by contradiction.
Case 1 (i is not selected): Let OPT be an optmal solution to $FD(i, w)$ but OPT is not optimal for $F(i-1, w)$. Then there exists another OPT' that is optimal for $F(i-1, w)$. However, OPT' will be a better solution for $FD(i, w)$ too.
Case 2 (i is selected): Let OPT be an optmal solution to $FD(i, w)$ but OPT$\backslash i$ is not optimal for $F(i-1, w-w_i)$. Then there exists another OPT' that is optimal for $F(i-1, w_i)$. However, $OPT' \cup i$ will be a better solution for $FD(i, w)$ too.

(b) (5 points) Please write down the recurrence relation you derived in (a), and use it to complete a DP table for solving the following problem instance.

Gift i	1	2	3	4
Weight w_i	2	2	4	3

Answer: The range of DP table are $0 \le i \le n$ and $0 \le w \le W$.

$$\begin{aligned} FD(0, w) &= w \text{ for all } w \\ FD(i, w) &= \min\{FD(i-1, w-w_i) - w_i, FD(i-1, w)\} \text{ if } FD(i-1, w-w_i) \ge w_i \\ &= FD(i-1, w) \text{ otherwise} \end{aligned}$$

$i \backslash w$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	0	1	0	1	2	3
2	0	1	0	1	0	1
3	0	1	0	1	0	1
4	0	1	0	0	0	0

(c) (5 points) You are now asked to divide n gifts into 2^k groups such that the sum of the pairwise weight differences (i.e., $\sum_{\forall i > j} |s_i - s_j|$, where s_i is the weight of group i) is minimized. k is a positive integer. You come up with a divide-and-conquer algorithm that recursively divides gifts into two groups while minimizing the weight difference of these two groups. The algorithm stops when the recursion depth reaches k.

Please provide a counterexample to show that this approach is not always optimal.

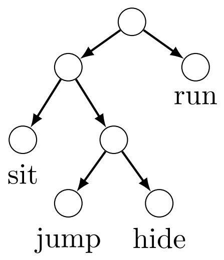
Answer: Divide {3 3 4 1 1 1 1 1 1 1 1 1 1} into four groups.
Step1: {3 3 4}, {1 1 1 1 1 1 1 1 1 1}
Step2: {3 3}, {4}, {1 1 1 1 1}, {1 1 1 1 1}
However, the best way to separate is {1 1 3}, {1 1 3}, {1 4}, {1 1 1 1 1}.

Problem 5: Zoologist (20 points)

As a zoologist, you're interested in studying how to communicate with Pokemons. Since most Pokemons can make sound, the length of sound can be used to encode 1-bit of information. We will use a long sound to represent 1 and a short sound to represent 0.

(a) (5 points) You want to teach Pokemons seven commands that are frequently used during training. Please draw an optimal prefix tree for encoding these commands:

Word	eat	sleep	jump	drink	run	sit	hide
Frequency	0.25	0.15	0.12	0.19	0.07	0.14	0.08



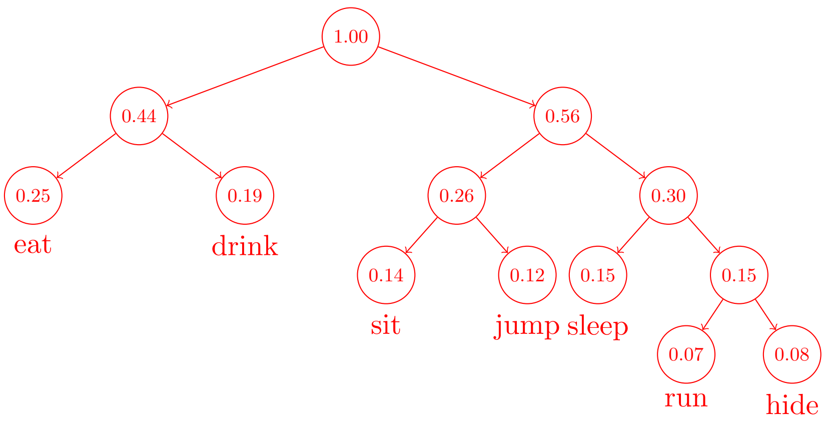
(b) (9 points) You found a secret training document written by another trainer, which shows that the trainer encodes four commands using a prefix tree as follows:

Suppose the prefix tree is constructed using Huffman encoding. For each of the following statements, please explain whether it is true or false:

- The command *run* must have a frequency higher than 1/3.
- The command *jump* must have a frequency less than 1/8.
- The frequency of command *run* is no less than it of command *sit*.

(c) (6 points) You observed that making a long sound consumes r times more energy than making a short one. Please revise the prefix code you came up with in (a) to minimize the expected energy consumption per word for the 7-word example when $r = 100$. Please briefly justify your answer.

Answer:



(a)

(b) (i) False ("higher" means $>$, not $\geq$).
(ii) False.
(iii) True.

(c) See the following table:

Word	eat	sleep	jump	drink	run	sit	hide
Frequency	0.25	0.15	0.12	0.19	0.07	0.14	0.08
Code	000000	01	0001	1	000001	001	00001

Problem 6

(a) (4% for algorithm correctness and time complexity, 6% for the proof of greedy choice property) Idea: Move as far as possible in each day, and rest the last safety city.

C style pseudo code:

```
start = 1;
while (l != n)
{
        // find the farthest city within 100 km
        r = start;
        while (r <= n and L[r] - L[start] <= 100) r += 1;

        // r is out of range, minus 1
        r -= 1;

        // find the safe city
        while (start < r and z[r] != 0) r -= 1;

        assert(start != r); // or no solution

        start = r;
        rest(start);
}
```

(You don't need to prove the time complexity of your algorithm, but you will get some penalties if your time complexity is incorrect.)

The following is the proof of the greedy choice property.

Proof. Let x is the city chosen by our greedy algorithm and OPT is the optimal solution.

If $x \in OPT$, it is done.

If $x \notin OPT$, assume y is the first rest choice city in OPT. From the greedy rule, we know that $L[y] \leq L[x]$. Let z is the second rest city in OPT. We have $L[z] - L[y] \leq 100$ and also $L[z] - L[x] \leq 100$. So the solution $OPT' = OPT \setminus \{y\} \cup \{x\}$ is also a valid solution and $|OPT| = |OPT'|$. □

(b) (4% for algorithm correctness and time complexity, 6% for the proof of the correctness of algorithm)

Let $f(x)$ is the optimal stress from city 1 to city x. Consider all city that can rest before x, we have

$$f(x) = \min_{\substack{1 \leq k < x \\ z[k]=0}} f(k) + (100 - (L[x] - L[k]))^2$$

$$f(1) = 0$$

(You don't need to prove the time complexity of your algorithm, but you will get some penalties if your time complexity is incorrect.)

Proof. Let $c(S)$ is the total stress of the solution S.

At the first, let OPT_i is the optimal solution of $f(i)$ and x is the last city rested in the OPT_i.

Suppose $OPT' = OPT_i \setminus \{i\}$ is not the optimal to $f(x)$. Let s is the solution of $f(x)$. We know that

$$c(s \cup \{i\}) = f(x) + (100 - (L[i] - L[x]))^2 < c(OPT') + (100 - (L[i] - L[x]))^2 = c(OPT_i)$$

Then we have a solution $s \cup \{i\}$ which is better than OPT_i. Contradiction.

Now, we know that OPT_i is optimal if all OPT_j are optimal for all $j < i$. We already prove the boundary case. ($f(1) = 0$), then the correctness is proved by induction. □

Problem 7

(a)

```
find_majority(A[1...n]):
  if n == 1:
    return A[1]

  el = find_majority(A[1...$\lfloor \frac{n}{2} \rfloor$])
  er = find_majority(A[$\lfloor \frac{n}{2} \rfloor$+1...n])
  if el == er:
    return el

  fl = frequency of el in A
  fr = frequency of er in A

  if fl > $\lceil \frac{n}{2} \rceil$:
    return el
  elif fr > $\lceil \frac{n}{2} \rceil$:
    return er
  else:
    reutrn NONE
```

(b)

S1:

```
find_majority(A[1...n]):
  original_A = A
  while A.size > 1:
    temp = []
    pair up the elements in A
    for each pair:
      if pair[2] == NONE or pair[1] == pair[2]:
        temp.append(pair[1])
    A = temp

  if A.size == 0 or frequency of A[1] in original_A <= $\lceil \frac{n}{2} \rceil$:
    return NONE
  else:
    return A[1]
```

S2: Moore's Voting Algorithm

```
find_majority(A[1...n]):
  maj_index = 0
  count = 0
  for i = 1 to n:
    if count == 0:
      maj_index = i
      count = 1
    elif A[maj_index] == A[i]:
      count++
    else:
      count--

if frequency of A[maj_index] in A > $\lceil \frac{n}{2} \rceil$:
  return A[maj_index]
else:
  return NONE
```