Exercise 0

Suppose you are investing. You want to buy low, then sell high. You have an array $A$ of $n$ integers representing future prices, and can make one buy followed by one sell. Regardless of the price at which you buy, you will only purchase a single unit. What is the maximum profit you can make on this investment?

(a) Design an $O(n \log n)$ divide-and-conquer algorithm to solve this problem, and justify its runtime.

(b) Design an $O(n)$ algorithm to solve this problem, and justify its runtime.

Solution 0

(a) We divide the array of prices in half. Then we can either buy and sell both in the first half, both in the second half, or buy in the first half and sell in the second half. We consider the first two cases using recursion, and calculate the maximum profit in the third case directly.

Algorithm 1: MaximumProfit(A)

```python
n = len(A)
if n = 1 then
    return 0
L = A[ : n/2], R = A[n/2 : ]
max_left = MaximumProfit(L)
max_right = MaximumProfit(R)
max_across = max(R) - min(L)
return max(max_left, max_right, max_across)
```

We divide the problem into two subproblems of size $n/2$ and perform $O(n)$ additional work, so our runtime is $T(n) = 2T(n/2) + O(n)$. This solves to $O(n \log n)$ via the Master Theorem.

(b) We do a single scan over the array and consider the maximum profit we could achieve by selling at each time step, which is achieved by buying at the minimum price seen up to that point.

Algorithm 2: MaximumProfit(A)

```python
n = len(A)
min_value = A[0]
max_profit = 0
for i = 1, \ldots, n-1 do
    max_profit = max(max_profit, A[i] - min_value)
    min_value = min(min_value, A[i])
return max_profit
```

This algorithm iterates through the array once, performing only constant-time operations in each iteration. All auxiliary work takes $O(1)$ time, so the total runtime is $O(n)$. 

1
Exercise 1

A string is a palindrome if it is the same both forwards and backwards. For example, “kayak” is a palindrome, but “canoe” is not. Similarly, “aa” is a palindrome, but “abaa” is not. (“a” is also a palindrome.)

A subsequence of a string is any sequence of characters that can be derived from the original string by deleting characters from that string. For example, the subsequences of the string “aid” are “aid”, “ai”, “ad”, “a”, “id”, “i”, “d”, and ”” (the empty string).

Design an algorithm that takes a string S and returns the length of the longest subsequence that is a palindrome. Analyze the runtime of your algorithm.

Solution 1

We approach this problem using dynamic programming.

Identify optimal substructure. It is natural to think of building a palindrome from both ends, so we will consider subproblems of the form \( f(i, j) \) representing the length of the longest palindromic subsequence in the substring \( S[i : j+1] \). Then we can compute the result for a substring by considering only the first and last characters and our pre-computed results for sub-substrings.

Namely, if the first and last characters match then the longest palindromic subsequence will include both; otherwise, we take the longest palindrome formed by either removing the first character or the last character.

Find a recursive formulation. More formally, we define

\[
 f(i, j) = \begin{cases} 
 2 + f(i + 1, j - 1) & \text{if } S[i] = S[j] \\
  \max(f(i + 1, j), f(i, j - 1)) & \text{else}
\end{cases}
\]

where \( f(i, i) = 1 \) and \( f(i, j) = 0 \) if \( i > j \).

Use dynamic programming. The following algorithm implements this recursive formulation using bottom-up dynamic programming. Note that our outer loop runs over increasing substring lengths, to ensure that the cells of \( A \) we look up at each step have already been computed.

**Algorithm 3: LongestSubsequencePalindrome(S)**

```python
n = len(S)
A = n by n array of 0's with 1's on the diagonal (at A[i][i])
for substringlen = 2, ..., n do
    for i = 0, ..., n - substringlen do
        j = i + substringlen - 1
        if S[i] = S[j] then
            A[i][j] = 2 + A[i+1][j-1]
        else
            A[i][j] = max(A[i+1][j], A[i][j-1])
    return A[0][n-1]
```

Runtime. The total number of sub-problems is \( O(n^2) \). The total time complexity per sub-problem is \( O(1) \), so the overall runtime of the algorithm is \( O(n^2) \).
Exercise 2

Suppose you want to start a petting zoo, and you’ve identified \( n \) animals you’d like to buy, each of which currently costs $100. However, it takes time to set up each exhibit, so you can only buy one animal each month. Additionally, for each month you wait to buy animal \( i \), its price goes up by a factor of \( r_i > 1 \) due to inflation, so if you buy animal \( i \) in month \( m \) it will cost \( 100 \times r_i^m \).

Design an algorithm which takes in the inflation rates \( r_i \) and returns the order in which you should buy animals to minimize your total cost, and prove that it is correct.

Solution 2

This problem has a greedy solution: always buy the animal with the highest inflation rate \( r_i \), because its price will increase the most if you wait. We prove the correctness of this algorithm inductively.

**Inductive hypothesis.** After choosing \( t \) animals, there exists an optimal solution extending our greedy solution.

**Base case.** After choosing 0 animals, any solution extends our solution.

**Inductive step.** Assume that after choosing \( t - 1 \) animals, there exists an optimal solution extending our greedy solution. Then our greedy algorithm will choose the animal with the maximum \( r_i \) to buy next. Suppose the optimal solution chooses a different animal \( j \) next and doesn’t buy animal \( i \) until \( m \) months later. We will construct a new optimal solution which buys animal \( i \) next.

Consider swapping animals \( i \) and \( j \) in the optimal solution. Then instead of paying \( 100 \times r_i^t + 100 \times r_i^{t+m} \) for the two animals, we would pay \( 100 \times r_j^t + 100 \times r_j^{t+m} \). (The cost of every other animal is unchanged.) Since \( r_i \geq r_j > 1 \), \( r_i^t + r_i^{t+m} \geq r_j^t + r_j^{t+m} \) so the modified solution is still optimal, i.e., there exists an optimal solution choosing animal \( i \) next.

**Conclusion.** After choosing \( n \) animals, there exists an optimal solution extending our greedy solution, i.e., our solution is optimal.

Exercise 3

Suppose that you’re playing a simplified version of minesweeper where there are \( n \) grid squares, and only one of them contains a mine. When you click on a square, it either reveals an empty slot, or reveals the mine and explodes. (No additional information is revealed, e.g., there are no numbers telling you how many adjacent slots contain mines, etc.) If you explode, you start over with a new board of size \( n \) and worst-case location of the mine. You decide to use a randomized algorithm—choosing uniformly at random from the remaining squares—to make your decisions about which square to click next. The game ends when you reveal all \( n - 1 \) safe slots on a single board without exploding.

(a) What is the probability of succeeding in one run without ever exploding?

(b) What is the expected number of squares visited until you win?

(c) How does your answer to (b) change if, after you explode, you are allowed to restart using the randomized algorithm at the point right before you exploded (e.g. if you explode with 10 slots (9 safe and 1 mine) left, your next move will be chosen uniformly at random from the same 10 slots)?

(d) Advanced (Take Home) - What if you only find out at the end of the game whether you exploded or succeeded, but if you exploded you can restart from any point in history?
Solution 3

(a) Considering each choice of slot sequentially, we have \( P[\text{success}] = \frac{n-1}{n} \times \frac{n-2}{n-1} \times \cdots \times \frac{3}{2} \times \frac{1}{2} = \frac{1}{n} \).

Alternatively, we note that every slot is equally likely to be chosen last by our algorithm (by symmetry), so the probability that the slot containing the mine is chosen last is \( \frac{1}{n} \).

(b) Using linearity of expectation, we can break this down as

\[
E[\# \text{ squares visited}] = \sum_{i=0}^{\infty} P[\text{don’t win before game } i] \times E[\# \text{ squares visited in game } i]
\]

\[
= E[\# \text{ squares visited per game}] \times \sum_{i=0}^{\infty} P[\text{don’t win before game } i]
\]

\[
= E[\# \text{ squares visited per game}] \times E[\# \text{ games played}],
\]

Because the mine slot is equally likely to be chosen 1st, 2nd, …, or nth in each game (but you never click all n squares), the first term is \( \frac{1}{n} (n-1+\sum_{i=1}^{n-1} i) = \frac{(n+2)(n-1)}{2n} \). By part (a) you win each game with probability \( 1/n \) so the second term is \( n \). Thus \( E[\# \text{ squares visited}] = \frac{(n+2)(n-1)}{2} = O(n^2) \).

(c) In this case, we save a lot of time because we don’t have to start over from scratch after every explosion. Instead, we can just add up the expected number of clicks before revealing the 1st, 2nd, …, and \((n-1)th\) empty slot:

\[
E[\# \text{ squares visited}] = \frac{n}{n-1} + \frac{n-1}{n-2} + \cdots + \frac{3}{2} + \frac{2}{1} < 2n = O(n).
\]

(d) Even if we don’t know exactly when we exploded, we can still save a lot by not starting over from scratch. In particular, the more squares you’ve already clicked, the more likely you are to mess up on the next square. By the time we’ve clicked \( n/2 \) squares (assuming we haven’t already failed), the probability of failing has doubled from \( 1/n \) for the first slot to \( 2/n \). Intuitively, this suggests that we should try the \( (n/2 + 1)th \) slot twice as often as the first slot.

Building on the above intuition (and ala Karger-Stein), consider the following algorithm: Try \( n/2 \) slots at random, then recurse twice on the remaining \( n/2 \) elements.

The number of slots visited by our algorithm is given by the following recurrence relation:

\[
T(n) = 2T(n/2) + n/2
\]

which solves to \( T(n) = O(n \log(n)) \) via the master theorem.

The probability that we succeed, i.e. visit all \( n-1 \) empty slots without exploding on at least one run, is given by

\[
P(n) = \frac{1}{2} \left( 1 - \left( 1 - P(n/2) \right)^2 \right).
\]

(Here the \( \frac{1}{2} \) factor is because we have probability 1/2 of exploding in the first \( n/2 \) slots, \( P(n/2) \) is the probability that we succeed when recursing on the remaining \( n/2 \) slots, and we square because we try the remaining \( n/2 \) twice.) Verify that this recurrence solves to \( P(n) = \Omega(\log(n)) \). Thus, we expect to have to repeat our algorithm \( O(\log(n)) \) times, for a total of

\[
O(\log(n)) \cdot T(n) = O(n \log^2(n))
\]

slot-visits.