
Advanced Algorithms

1. (a) T
- (b) F
- (c) T
- (d) It's only fixed-parameter tractable. As we increase the size of W (number of bits required to represent it), the runtime doubles. Thus, our algorithm is actually exponential with respect to the input.
- (e) When analyzing algorithms with multiple input parameters, we want to describe the efficiency of the algorithm with respect to a subset of the overall parameters (often a single one). An example in which this type of analysis was useful was for 0/1 Knapsack.
- (f) A decision version of the SSSP optimization problem might be "Does there exist a shortest path from s to t of at most cost c ?" We can use a solver for this polynomial-time decision problem by simply doing a open-ended binary search for the correct value c .

Final Review

1. (a) In words, our algorithm does the following:
 - Pick an arbitrary vertex $v \in V$.
 - Run BFS in G from v and check that all of V is reachable from v ; if not, return "no."
 - Construct the reverse graph $G' = (V, E')$ where $E' = \{(v, u) : (u, v) \in E\}$.
 - Run BFS in G' from v and check that all of V is reachable from v ; if not, return "no" and otherwise, return "yes."
- (b) For each edge $e \in E$ run the algorithm of part (a) on $G - \{e\}$ the graph obtained by removing e . Return "no" if any of these returns "no"; otherwise return "yes."
- (c) The minimum number of pipes/edges is $2n$.
 - It is necessary that for every vertex $v \in V$ to have in-degree of at least 2—that is, at least 2 edges pointing towards it. If not, then if we removed the sole edge $(u, v) \in E$ from G , then v cannot reach any other vertices. The same argument shows that we require an out-degree of at least to 2. If this is the case, then the number of edges is

$$|E| = \frac{1}{2} \sum_{v \in V} \deg(v) \geq \frac{1}{2} \sum_{v \in V} 4 \geq 2n$$

where $1/2$ comes from the fact that each edge contributes to the degree of two vertices.

- The condition above is also sufficient. To see this, number the vertices v_1, \dots, v_n and consider the "double-cycle" graph containing all edges (v_i, v_{i+1}) and (v_{i+1}, v_i) . This graph remains strongly connected if any single edge is removed, and it has exactly $2n$ edges.

2. (a) $(C, F), (E, F), (D, E), (D, A), (A, B)$
 (b) $(A, D), (A, B), (D, E), (C, F), (E, F)$
3. (a) Consider the different components of the recursive relationship

$$D[i] = \text{max}(\{D[k] + 1 : 0 \leq k < i, A[k] < A[i]\} \cup \{1\})$$

- The **max** operator indicates we're setting $D[i]$ to the maximum of a set of numbers.
 - The $D[k] + 1 : 0 \leq k < i$ means that we add 1 to values in the table to the left of the i and include this value as part of the set if appending element i to the end of a subsequence ending at element k satisfies the constraint $A[k] < A[i]$ i.e. the sequence will still be increasing.
 - The **1** accounts for the situation in which the previous set is empty, which could happen if element i happens to be smaller than all elements to the left of it. In this case, there still exists a trivial increasing subsequence ending at element i , namely the subsequence with element i itself.
- (b) Our pseudocode works as follows:

```
def lis(A):
    D = [0 for _ in range(len(A))]
    for i in range(len(A)):
        maxSoFar = 1
        for k in range(i):
            if A[k] < A[i] and D[k] + 1 > maxSoFar:
                maxSoFar = D[k] + 1
        D[i] = maxSoFar
    return max(D)
```

This algorithm runs in $O(n^2)$ -time since it iterates over all n elements of A and, for each element i , iterates over i elements to left in D to determine the maximum so far.

- (c) In addition to maintaining the lengths in the table, we can maintain the subsequences themselves. In addition to adding 1 to $D[k]$ to produce the maximum such $D[i]$, we append i to that maximum prior set chosen.
4. (a) We proceed by induction on the non-negative integer n .
 - **Inductive Hypothesis:** `minimumElements(i, S)` returns the correct solution for $0 \leq i$.
 - **Base Case:** We prove the inductive hypothesis for $i = 0$. Notice that 0 elements from S are required to write 0 as the sum of the elements, and the algorithm correctly returns 0. For all other $i < \min S$, it's impossible to write i as the sum of elements from S since $i < s_j < s_j + \sum_{s \in \bar{S}} s$ for all j and non-empty subsets of S given by \bar{S} . Thus, the algorithm correctly returns None in these cases.
 - **Inductive Step:** Suppose that `minimumElements(i, S)` returns the correct solution for all $0 \leq i < k$, where $k > \min S$. Here, we prove that it returns the correct solution for k . The algorithm retrieves the minimum number of elements needed to write $i = k - s$ for all $s \in S$. We know the algorithm returns the correct solutions to these problems by the inductive hypothesis. For each s , it's possible to write k as the sum of the same set of numbers from S needed to write i , and then including one more s . Thus, the minimum number of elements needed to write k is just the minimum value of all possible $i = k - s$, plus 1. Together, this proves the algorithm returns the correct solution for k .
 - **Conclusion:** The algorithm returns the correct solution for all i , including $i = n$, the number it was original called on. Therefore, `minimumElements(n, S)` returns the correct solution.

- (b) We show that the runtime of the algorithm takes the form $2^{\Omega(n)}$ for $S = \{1, 2\}$.
Consider the recurrence relation for the algorithm for this particular S :

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n < \min S \\ T(n-1) + T(n-2) + O(1) & \text{otherwise} \end{cases}$$

Since $T(n) = T(n-1) + T(n-2) + O(S) \geq 2T(n-2) + O(1)$, and expression on the right-hand side produces a recursion tree with $(n - \min S)/2 + 1$ levels and $2^{(n - \min S)/2 + 1} - 1$ total nodes. Since $\min S$ is a constant with respect to n , then $T(n) = T(n-1) + T(n-2) \geq 2^{\Omega(n)}$, as required.

- (c) In words, we just add a cache to keep track of solutions already solved.

```
# Use an array instead of a dict for O(1)-time worst-case search
cache = [-1 for _ in range(n)]
def minimumElements(n, S):
    if n == 0:
        return 0
    if n < min(S):
        return None
    if cache[n] != -1:
        return cache[n]
    candidates = []
    for s in S:
        candidate = minimumElements(n-s, S)
        if candidate is not None:
            candidates.append(candidate + 1)
    if len(candidates) == 0:
        minCandidate = None
    else:
        minCandidate = min(candidates)
    cache[n] = minCandidate
    return minCandidate
```

This algorithm recursively solves for the minimum candidate at most once for all $0 \leq i < n$ and just indexes into the cache in worst-case $O(1)$ -time otherwise. Each of the n times the algorithm actually solves for the minimum candidate, it iterates over a list of length $|S|$. Therefore, the algorithm runs in worst-case $O(n|S|)$ -time.

- (d) In words, we fill out a table associated with the solutions for $i = \{0, \dots, n\}$.

```
table = [-1 for _ in range(n+1)]
def minimumElements(n, S):
    table[0] = 0
    for i in range(1, min(S)):
        table[i] = None
    for k in range(min(S), n+1):
        for s in S:
            candidate = minimumElements(k-s, S)
            if candidate is not None:
                candidates.append(candidate + 1)
        if len(candidates) == 0:
            minCandidate = None
        else:
            minCandidate = min(candidates)
        table[k] = minCandidate
    return table[-1]
```

5. (a) Your friend is incorrect. Consider $n = 30$. The greedy algorithm would find the solution $\{1, 1, 1, 1, 1, 25\}$, but the optimal solution is $\{10, 10, 10\}$.
- (b) Your friend is correct.

First, we prove a lemma: For all s , any list comprised of the elements $\{1, 2, 4, 8, \dots, 2^s\}$ that sums to at least 2^s (each element might appear in the list multiple times) must contain a sublist of elements that sums to exactly 2^s .

To prove the statement, we proceed by induction.

- **Inductive Hypothesis:** A list comprised of the elements $\{1, 2, 4, 8, \dots, 2^s\}$ that sums to at least 2^s must contain a sublist of elements that sums to exactly 2^s .
- **Base Case:** The statement holds for $s = 0$: for any list comprised of just 1's that sums to at least 1, the list contains a subset of elements (namely, any one of the elements!) that sums to exactly 1.
- **Inductive Step:** Suppose the inductive hypothesis holds for s ; we prove it holds for $s + 1$. Consider a list comprised of the elements $\{1, 2, 4, 8, \dots, 2^s, 2^{s+1}\}$ that sums to at least 2^{s+1} . Let $x \geq 2^{s+1}$ be this sum. Consider two cases on this list:
 1. It contains at least one occurrence of the element 2^{s+1} . There exists a trivial sublist of elements containing this single element that sums to exactly 2^{s+1} , completing the induction.
 2. It does not contain any occurrences of the element 2^{s+1} , and entirely contains elements from $\{1, 2, 4, 8, \dots, 2^s\}$. By the inductive hypothesis, since $x \geq 2^{s+1} > 2^s$, there exists a sublist of elements that sums to exactly 2^s . Excluding these elements, the remaining elements sum to $x - 2^s \geq 2^{s+1} - 2^s \geq 2^s$. Therefore, by the inductive hypothesis, there exists another sublist (disjoint from the first) that sums to exactly 2^s . Since there are two sublists that sum to exactly 2^s , there must be a sublist that sums to exactly 2^{s+1} , namely the concatenation of these two sublists, completing the induction.
- **Conclusion:** Since the inductive hypothesis holds for $s = 0$ (base case), and it holds for $s + 1$ if it holds for s (inductive step), it holds for all s , as required.

Using the lemma, we can prove that the greedy algorithm is always feasible and optimal for this specific P .

First, notice the algorithm is feasible since it never makes too much change (by constraining $p_{\text{opt}} \leq n$) and it never makes too little change since it only halts when $n = 0$.

In addition, we argue that setting p_{opt} to $p_{\text{max}} = \max\{p \in P : p \leq n\}$ is optimal. In the direction of contradiction, assume it's actually optimal to *omit* this maximum element $p_{\text{max}} = 2^s \leq n$ from the coins. (Note that p_{max} can be written as a power of two since all elements of P can be written in this form.) This optimal solution must sum to $n \geq 2^s$, using only the elements $\{1, 2, 4, 8, \dots, 2^{s-1}\}$ since using any elements larger than p_{max} would make too much change by construction of p_{max} . By the lemma, a sublist of this optimal solution must sum to exactly 2^s . Since the optimal solution does not contain p_{max} , this sublist must contain strictly more than one item. Replacing this sublist with p_{max} still sums to n , but uses fewer coins, contradicting the optimality of the solution.

Therefore, our assumption must have been incorrect, and the optimal solution must contain p_{max} , proving the optimality of the greedy solution.