Algorithms & Data Structures I

Techniques, Correctness & Complexity (Courses 1 → 12)

1. Brute Force Technique

Definition. Directly follows the problem definition and explores all possibilities or performs all required steps explicitly.

Canonical exam problem. Compute n! (factorial of a natural number n).

Python 3 solution:

```
def factorial(n):
result = 1
for i in range(1, n + 1):
result *= i
return result
```

Correctness. Loop invariant: after i iterations, result = i!. Termination gives result = n!.
Time complexity. Θ(n)

2. Reduction Technique (Decrease & Conquer)

Definition. Reduce the problem to a smaller instance of the same problem, solve it, then extend the solution.

Canonical exam problem. Fast exponentiation x^n.

Python 3 solution:

```
def power(x, n):
if n == 0:
return 1
p = power(x, n // 2)
if n % 2 == 0:
return p * p
return p * p * x
```

Correctness. Correct by induction on n using the recursive definition of exponentiation.
Time complexity. Θ(log n)
Remarks. Uses recurrence T(n)=T(n/2)+1.

3. Divide & Conquer

Definition. Divide the problem into independent subproblems of smaller size, solve each recursively, then combine the results.

Canonical exam problem. Binary search in a sorted array.

Python 3 solution:

```
def binary_search(a, x):
left, right = 0, len(a) - 1
while left <= right:
mid = (left + right) // 2
if a[mid] == x:
return True
```

```
if x < a[mid]:
right = mid - 1
else:
left = mid + 1
return False
```

Correctness. Invariant: if x exists, it is always in the interval [left, right].
Time complexity. Θ(log n)
Remarks. Only one subproblem is solved at each step (reduction).

4. Recursive Algorithms

Definition. Solve a problem by calling the same algorithm on smaller inputs until a base case is reached.

Canonical exam problem. Recursive Fibonacci computation.

Python 3 solution:
```
def fib(n):
if n <= 2:
return 1
return fib(n - 1) + fib(n - 2)
```

Correctness. Correct by induction on n.
Time complexity. Θ(ϕ^n)
Remarks. Illustrates inefficiency without memoization.

5. Elementary Sorting – Insertion Sort

Definition. Incrementally builds a sorted array by inserting each element into its correct position.

Canonical exam problem. Sort an array using insertion sort.

Python 3 solution:
```
def insertion_sort(a):
for i in range(1, len(a)):
key = a[i]
j = i - 1
while j >= 0 and a[j] > key:
a[j + 1] = a[j]
j -= 1
a[j + 1] = key
return a
```

Correctness. Invariant: after iteration i, subarray a[0..i] is sorted.
Time complexity. Θ(n²) worst-case, Θ(n) best-case
Remarks. Stable and in-place.

6. Greedy Technique

Definition. Builds a solution step by step by always choosing the locally optimal option without backtracking.

Canonical exam problem. Activity selection problem.

Python 3 solution:
```
def activity_selection(starts, ends):
acts = sorted(zip(starts, ends), key=lambda x: x[1])
```

```
result = []
last_end = -1
for s, e in acts:
if s >= last_end:
result.append((s, e))
last_end = e
return result
```

Correctness. Correct due to greedy-choice property and optimal substructure.
Time complexity. Θ(n log n) due to sorting
Remarks. Does not always guarantee optimality for all problems.

7. Dynamic Programming

Definition. Solves problems with overlapping subproblems and optimal substructure by storing intermediate results.

Canonical exam problem. Edit distance between two strings.

Python 3 solution:
```
def edit_distance(x, y):
m, n = len(x), len(y)
dp = [[0]*(n+1) for _ in range(m+1)]
for i in range(m+1): dp[i][0] = i
for j in range(n+1): dp[0][j] = j
for i in range(1, m+1):
for j in range(1, n+1):
if x[i-1] == y[j-1]:
dp[i][j] = dp[i-1][j-1]
else:
dp[i][j] = 1 + min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1])
return dp[m][n]
```

Correctness. Table invariants ensure dp[i][j] is optimal for prefixes x[1..i], y[1..j].
Time complexity. Θ(m·n)
Remarks. Bottom-up approach.

8. Backtracking

Definition. Explores the solution space incrementally and abandons partial solutions that violate constraints.

Canonical exam problem. Generate all permutations of a list.

Python 3 solution:
```
def permutations(a, pos=0):
if pos == len(a):
print(a)
return
for i in range(pos, len(a)):
a[pos], a[i] = a[i], a[pos]
permutations(a, pos + 1)
a[pos], a[i] = a[i], a[pos]
```

Correctness. Correct because all valid configurations are explored exactly once.
Time complexity. Θ(n!)
Remarks. Worst-case exponential complexity.