Tutorial 7: Common Algorithmic Techniques

The Art of Computer Programming

4 November 2026

Objective. This tutorial puts into practice the seventh lecture: the same recurrence evaluated in three ways (recursion, memoization, a table) on the making-change problem, with timings; the fewest coins, which coins they are, and a greedy rule that sometimes fails; the edit distance and the sequence of edits behind it; and the master theorem on a few short programs. Everything is in Python today. Do Parts A and B in the first half of the session and Parts C and D in the second half; Part E is for those who finish early.

Reminders

- Create a folder `tutorial07` with two files, `change.py` (Parts A, B, and E) and `editdistance.py` (Part C).
- `time.perf_counter()` gives a time in seconds (tutorial 2). Time each call once, with inputs growing until a call takes about a second, and stop there: do not wait for minutes.
- Python raises `RecursionError` beyond about 1 000 nested calls. This is expected in one question; do not raise the limit, change the algorithm.
- `float("inf")` is $+\infty$: it is larger than every number, and `float("inf") + 1` is still $+\infty$.
- A table of r rows and c columns of zeros is `[[0] * c for _ in range(r)]`, never `[[0] * c] * r` (all the rows would be the same list, lecture 3).

Part A: Making change, three ways

Coins of values $p_1 \le \cdots \le p_n$ in unlimited supply, a sum S, and the recurrence of the lecture for the number of ways $N(S, k)$ of making S with the first k kinds of coins:

$$N(0,k) = 1, \qquad N(S,0) = 0 \text{ for } S > 0, \quad N(S,k) = 0 \text{ for } S < 0,$$
$$N(S,k) = N(S,k-1) + N(S-p_k,k) \text{ for } S,k > 0.$$

In `change.py`, define the tuple `EURO` of the eight euro coins, `(1, 2, 5, 10, 20, 50, 100, 200)`.

Q1. Recursion.

(a) Write `nb_ways_rec(s, p)`, which returns $N(s,n)$ by calling an inner recursive function `n(s, k)` that follows the recurrence line by line. Check: 7 with coins $(1,2,5)$ gives 6, 10 with the same coins gives 10, and 0 with the euro coins gives 1.

(b) Time it with the euro coins for $S = 50$, 100, 200, and 300. By roughly what factor does the time grow each time S increases by 100? Without running it, is $S = 1\,000$ within reach?

Q2. Memoization.

(a) Copy the function as `nb_ways_memo(s, p)` and add a memo, a dictionary from pairs `(s, k)` to results, as in the lecture: look it up first, store the result before returning. Check the same values, then time $S = 300$, 500, and 700: compare with Q1.

(b) After the call for $S = 300$, print the number of entries of the memo and compare it with the number of pairs (S,k) of the table, $(S+1) \times (n+1)$. Why are there fewer?

(c) Run it for $S = 1\,000$. What happens, and why? (Which chain of calls is the longest?)

Q3. A table.

(a) Write `nb_ways_table(s, p)`, which fills a table `t` of $(s+1)$ rows and $(n+1)$ columns such that `t[i][k]` is $N(i,k)$, in the order of the lecture (every sum, then every number of kinds of coins), and returns `t[s][n]`. Check the same values, then $S = 1\,000$ (321 335 886 ways) and time $S = 10\,000$ and 100 000.

(b) How do the time and the memory grow with S? With n? Why is there no recursion limit to fear here?

Part B: The fewest coins, and a greedy rule

The lecture's recurrence for the smallest number of coins $M(S,k)$, with $+\infty$ meaning "impossible":

$$M(0,k)=0, \qquad M(S,0)=+\infty \text{ for } S>0, \qquad M(S,k)=\min\big(M(S,k-1),\ M(S-p_k,k)+1\big) \text{ for } S,k>0,$$

where the second term is only taken when $p_k \le S$.

Q4. Fewest coins.

(a) Write `min_coins(s, p)`, a table like Q3 with `float("inf")` in place of the impossible cases, returning $M(s,n)$. Check: 137 cents in euro coins take 5; 6 with coins $(1,3,4)$ takes 2; 7 with coins $(2,5)$ takes 2; 3 with coins $(2,5)$ is impossible.

(b) Make it return the coins as well: keep a second table `used[i][k]`, `True` when the minimum for (i,k) was reached by taking a coin p_k, then walk back from (s,n): if `used[i][k]`, take the coin and subtract it from i, otherwise decrease k; stop when $i=0$. For 137 cents you should get `[100, 20, 10, 5, 2]` (in some order).

(c) The table contains $+\infty$ in places. Why do those entries never produce a wrong answer?

Q5. Greedy.

(a) Write `greedy(s, p)`, which returns the list of coins obtained by repeatedly taking the largest coin that fits. Check it on 137 cents.

(b) For every S from 1 to 300, compare the number of coins of `greedy(S, EURO)` with `min_coins(S, EURO)`. Do they ever differ?

(c) Same comparison with the US coins of 1875, $p=(1,5,10,20,25)$: print the smallest S where they differ, and both solutions. Then with $p=(1,10,25)$.

(d) Why can the table of Q4 never be wrong, whatever the coins, where the greedy rule can?

Part C: Edit distance

In `editdistance.py`, for two strings s and t, $P(i,j)$ is the edit distance between the first i characters of s and the first j of t:

$$P(i,0)=i, \qquad P(0,j)=j, \qquad P(i,j)=\min\big(P(i-1,j)+1,\ P(i,j-1)+1,\ P(i-1,j-1)+[s_i \neq t_j]\big) \text{ for } i,j\ge 1,$$

where $[s_i \neq t_j]$ is 1 if the characters differ and 0 otherwise (in Python, `s[i-1] != t[j-1]` used as a number).

Q6. Three ways again.

(a) Write `distance_rec(s, t)` with an inner recursive function `d(i, j)`. Check: `"chien"` to `"chat"` is 3, `"kitten"` to `"sitting"` is 3, `""` to `"abc"` is 3, and a word to itself is 0.

(b) Time it on `"a" * n` and `"b" * n` for $n=6$, 8, and 10. Each time n grows by 2, by what factor does the time grow?

(c) Write `distance_memo(s, t)`, memoized with a dictionary, and time $n=10$, 100, and 300. What would happen for $n=600$, and why?

(d) Write `distance_table(s, t)`, filling a table `d` of $(len(s)+1)\times(len(t)+1)$ entries row by row, and time $n=500$ and 1 000.

Q7. The edits. Make `distance_table` also return the list of edits that turn s into t, by walking back from the bottom right cell $(i,j)=(len(s),len(t))$ to $(0,0)$: at each step, find which of the three terms of the recurrence gave `d[i][j]` (when $i=0$ or $j=0$, only insertions or only deletions remain), record the corresponding edit (keep, substitute, delete, insert), and move to the corresponding cell. The edits are found last to first: reverse the list at the end. For `"chien"` to `"chat"` you should get: keep c, keep h, delete i, substitute e by a, substitute n by t (another sequence of three edits is possible, depending on which term you test first); check `"kitten"` to `"sitting"` too.

Part D: The master theorem, on paper

Q8. Four functions. Each of the functions below works on the elements `t[lo]` to `t[hi-1]` of a list, $n = hi - lo$ a power of 2, and is called as `f(t, 0, len(t))`. For each one, write its recurrence $T(n) = a\,T(n/b) + f(n)$, give $c = \log_b a$, say which case of the master theorem applies, and conclude.

Python

```
def total(t, lo, hi):       # (a)
    if hi - lo == 1:
        return t[lo]
    m = (lo + hi) // 2
    return total(t, lo, m) + total(t, m, hi)

def spread(t, lo, hi):      # (b)
    if hi - lo == 1:
        return 0
    m = (lo + hi) // 2
    s = spread(t, lo, m) + spread(t, m, hi)
    for i in range(lo, hi):
        s += t[i]
    return s
```

Python

```
def quad(t, lo, hi):        # (c)
    if hi - lo == 1:
        return t[lo]
    m = (lo + hi) // 2
    a = quad(t, lo, m) + quad(t, m, hi)
    b = quad(t, lo, m) + quad(t, m, hi)
    return a + b

def half(t, lo, hi):        # (d)
    if hi - lo == 1:
        return t[lo]
    m = (lo + hi) // 2
    s = 0
    for i in range(lo, hi):
        s += t[i]
    return s + half(t, lo, m)
```

Then check (c) by timing it on lists of 2^{10}, 2^{11}, and 2^{12} elements: by what factor does the time grow when n doubles, and is this what the theorem predicts?

Part E: If you have time

Q9. The 0–1 knapsack. In `change.py`, write `knapsack(objects, capacity)`, where `objects` is a list of pairs (weight, value), using the recurrence of the lecture for $V(c, k)$, the best value with capacity c and the first k objects, in a table of $(capacity + 1) \times (n + 1)$ entries. Make it return the chosen objects too, by walking the table back as in Q4(b). On the example of the lecture, objects $(10, 60)$, $(20, 110)$, $(30, 150)$ and capacity 50, the value is 260 with the second and third objects. Compare with the greedy rule "by decreasing value per unit of weight": which value does it reach?

Check

By the end of the session you should be comfortable with: turning a recurrence into a recursive function, then into a memoized one, then into a table, and knowing what each one costs in time, memory, and recursion depth; reconstructing a solution by remembering the choice that achieved each minimum; checking a greedy rule against the exact answer; and reading a, b, and f off a recursive function to apply the master theorem. If one of them is still unclear, ask us, or finish the corresponding exercises at home.