Lecture 14

Dynamic programming III

Chinmay Nirkhe | CSE 421 Winter 2026

Previously in CSE 421…

Knapsack runtime

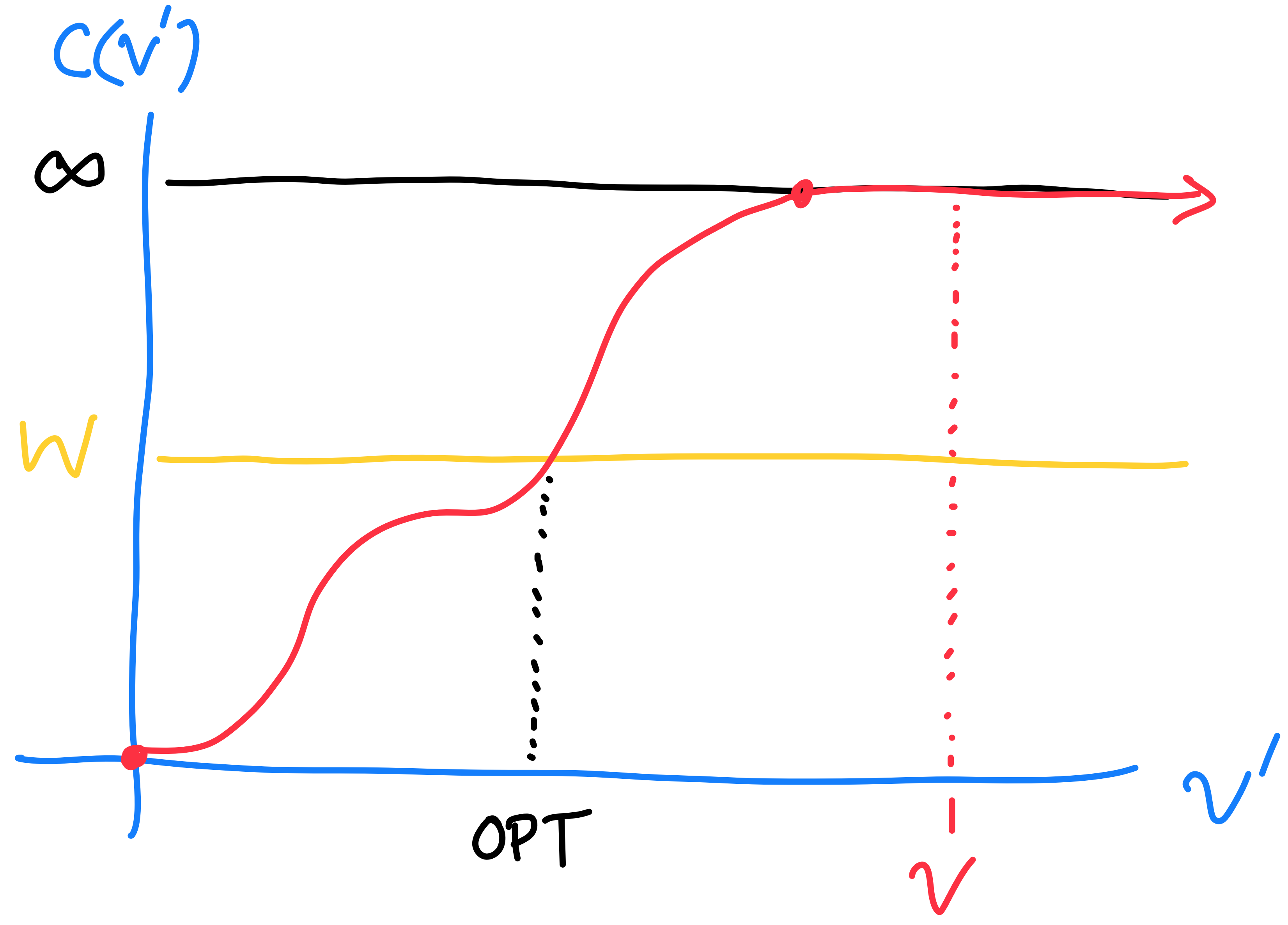
- The input for Knapsack is usually written in **binary** with each item weight w_i expressed with $O(\log W)$ bit numbers and value with $O(\log V)$ bit numbers
- Total input length is $\Theta(n \log V + n \log W) = \Theta(n \log VW)$
- Runtime of Knapsack brute-force alg is $O(n2^n \log VW)$, exp in input length
- Runtime of Knapsack DP alg is $O(nW \log VW)$ also exp in the input length
- **DP algorithm is only faster when $W \ll 2^n$.**

Knapsack approximation algorithm

- Given a Knapsack problem $(v_1, \ldots, v_n, w_1, \ldots, w_n, W)$, let OPT be the optimal value of subset of items weighing $\leq W$: $\mathrm{OPT} = V(n, W)$
- An alg. $\mathscr{A}$ is an **ϵ-approximation alg.** if $\mathscr{A}$ always outputs a subset $\tilde{S}$ such that (a) $\mathrm{weight}(\tilde{S}) \leq W$ and (b) $\mathrm{value}(\tilde{S}) \geq (1-\epsilon) \cdot \mathrm{OPT}$.
- **Theorem:** For every $\epsilon > 0$, there exists an ϵ-approximation alg. for n-item Knapsack that runs in time $O\left(\frac{n^3 \log(VW)}{\epsilon}\right)$.
- The construction will be another dynamic programming algorithm.

A different DP algorithm for (exact) Knapsack

- Assume that $0 \le w_i \le W$ for all items.
- Let $v_{\max} = \max_i v_i$. Then, $v_{\max} \le \text{OPT} \le V$
- **Define**: $C(V')$ to be the minimum weight of a set S such that $\text{value}(S) \ge V'$
    - Let $C(V') = \infty$ if no set S exists of this value.
    - Base case of $C(0) = 0$
    - $C(V') = \infty$ for $V' > V$
    - $C(V')$ is monotonically increasing
- Then, Knapsack solution OPT = max value V' s.t. $C(V') \le W$



A slightly different optimization

- $C(V')$ can be "morally" seen as a dual problem to maximization $V(W')$
- **Define:** $C(i, V')$ as the minimum weight of a set S such that $\mathrm{value}(S) \geq V'$ using items only $\{1, \ldots, i\}$
  - This new subproblem has a recursive definition similar to our previous example
  - $C(i, V') = \min \left\{ \begin{array}{c} C(i-1, V'), \\ C(i-1, V' - v_i) + w_i \end{array} \right\}$
- The table $C(\cdot, \cdot)$ consists of $O(nV)$ entries
- **Observe** $C(V') = C(n, V')$
- **Observe** OPT = the maximum value V' s.t. $C(n, V') \leq W$

A different Knapsack algorithm

- This new algorithm has a table of size $(n + 1) \times V$
- Each entry of the table can be constructed in $O(\log W + \log V) = O(\log VW)$ time
- Computing OPT after table involves binary searching along $C(n, \cdot\,)$ as $C(n, \cdot\,)$ is monotonic
  - OPT = the maximum value V' s.t. $C(n, V') \leq W$
  - Requires $O(\log V(\log VW))$ total compute
- **Yields a total runtime of** $O(nV \log VW)$
  - No exponential dependence in terms of $\log W$
  - However, exponential dependence in terms of $\log V$

An approximation algorithm

- **Yields a total runtime of** $O(nV \log VW)$
- What if we just replaced each v_i with v_i/Z for a large number Z?
  - Would the algorithm now run in $\tilde{O}\left(\frac{nV \log VW}{Z}\right)$ as the sum of values is now V/Z?
  - **No**. Crucially, to run the dynamic programming algorithm we needed all the values to be **integers**.
- However, this suggests an *approximation algorithm*.
- **Approximation algorithm (overview):**
  - Define $\tilde{v}_i := \lfloor v_i/Z \rfloor$. Return $S \leftarrow \text{Knapsack}(\{\tilde{v}_i\}, \{w_i\}, W)$ with our *second* DP algorithm.

An approximation algorithm

- **Idea**: Compute $S \leftarrow \text{Knapsack}(\{\tilde{v}_i\}, \{w_i\}, W)$ for $\tilde{v}_i = \lfloor \frac{v_i}{Z} \rfloor$ & $Z = \frac{\epsilon v_{\max}}{n}$.

  Since the weights are reduced, the runtime is shorter!

- **Runtime**: $O\left(\frac{nV \log VW}{Z}\right) = O\left(\frac{n^2 V \log VW}{\epsilon v_{\max}}\right) \leq O\left(\frac{n^3 \log VW}{\epsilon}\right)$

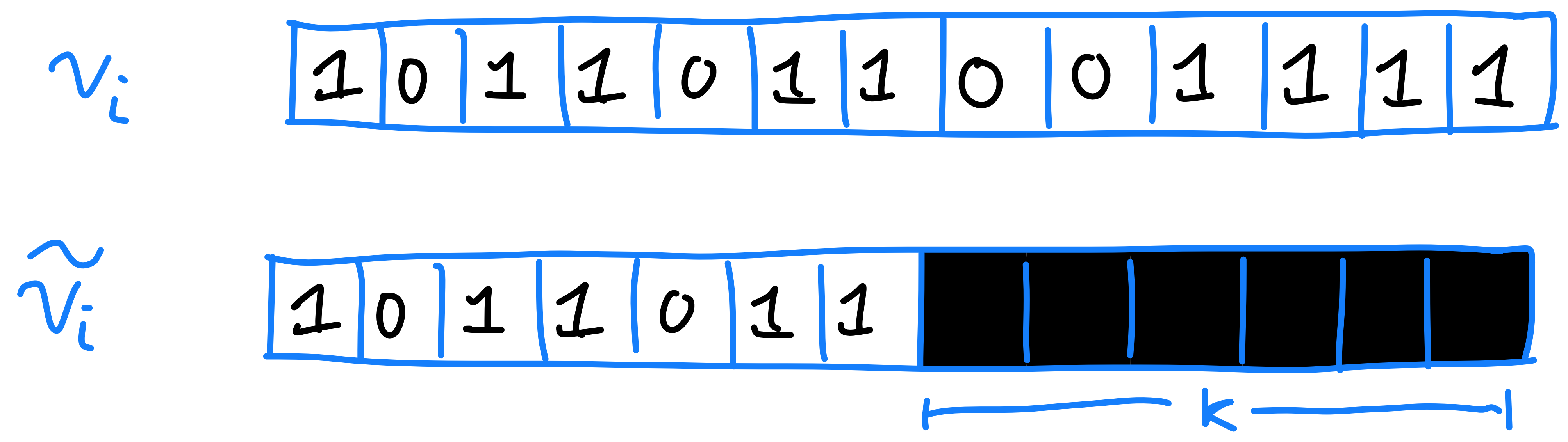
- **Claim**: S is a feasible solution and $\text{value}(S) \geq (1-\epsilon)\text{OPT}$.

An approximation algorithm

- **Idea:** Compute $S \leftarrow \text{Knapsack}(\{\tilde{v}_i\}, \{w_i\}, W)$ for $\tilde{v}_i = \lfloor \frac{v_i}{Z} \rfloor$ & $Z = \frac{\epsilon v_{\max}}{n}$.
    - Since the weights are reduced, the runtime is shorter!

For intuition, say $Z = 2^k$ for some k.

Then if we express v_i is binary:



Keep only the significant digits. This alg. is morally rounding.

An approximation algorithm

Let $\tilde{v}_i = \left\lfloor \frac{v_i}{Z} \right\rfloor$ for $Z = \frac{\epsilon v_{max}}{n}$. Output $S \leftarrow \text{Knapsack}(\{\tilde{v}_i\}, \{w_i\}, W)$.

Claim: S is a feasible solution to the original problem.

Proof: Since the weights $\{w_i\}$ and limit W are the same in both problems,

then $\sum_{i \in S} w_i \leq W$.

An approximation algorithm

Let $\tilde{v}_i = \left\lfloor \frac{v_i}{Z} \right\rfloor$ for $Z = \frac{\epsilon v_{max}}{n}$. Output $S \leftarrow \text{Knapsack}(\{\tilde{v}_i\}, \{w_i\}, W)$.

Let $\text{value}(S) = \sum_{i \in S} v_i$, $\widetilde{\text{value}}(S) = \sum_{i \in S} \tilde{v}_i$.

Let O be the optimal sol. to $\text{Knapsack}(\{v_i\}, \{w_i\}, W)$.

So, $\text{OPT} = \text{value}(O)$.

Claim: $\text{value}(S) \geq (1-\epsilon)\,\text{OPT}$.

An approximation algorithm

Claim: $\text{value}(S) \geq (1-\epsilon)\,OPT$.

since $Z = \frac{\epsilon V_{max}}{n}$

Proof: For any item i, $v_i - Z\tilde{v}_i = Z\left(\frac{v_i}{Z} - \left\lfloor \frac{v_i}{Z} \right\rfloor\right) \leq Z$.

Since O has $\leq n$ items, $OPT - Z\,\widetilde{\text{value}}(O) = \sum_{i \in O} v_i - Z\tilde{v}_i \leq nZ = \epsilon V_{max}$

$$Z\,\widetilde{\text{value}}(O) \geq OPT - \epsilon V_{max} \geq (1-\epsilon)\,OPT. \quad (1)$$

Next, $\widetilde{\text{value}}(S) \underset{(2)}{\geq} \widetilde{\text{value}}(O)$ since S is optimal sol. to Knapsack($\{\tilde{v}_i\}, \{w_i\}, W$)

So, $\text{value}(S) \geq Z\,\widetilde{\text{value}}(S) \underset{(2)}{\geq} Z\,\widetilde{\text{value}}(O) \underset{(1)}{\geq} (1-\epsilon)\,OPT$. ∎

Structure of approx. DP algorithm

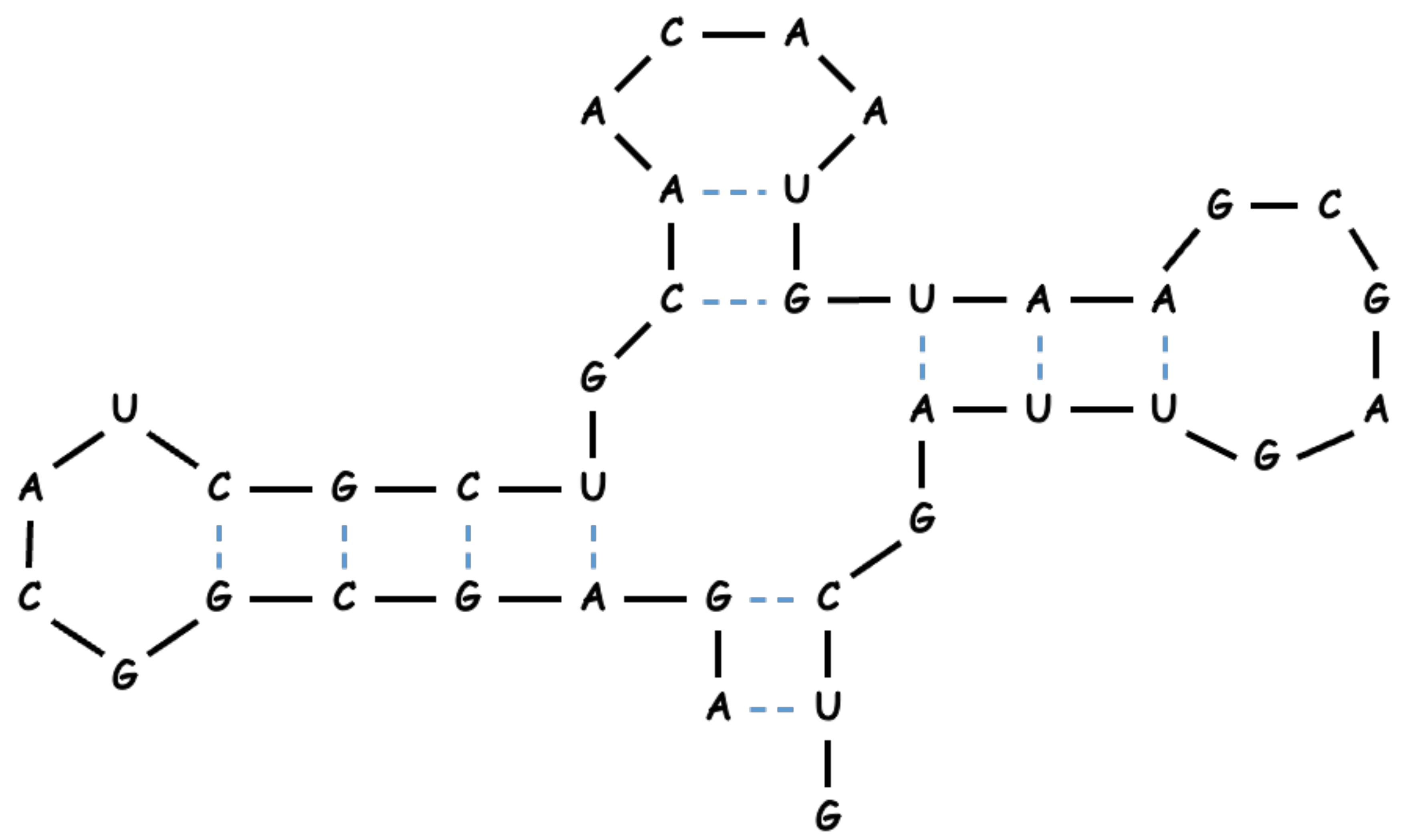
- We came up with two DP algorithms for **exact** Knapsack based on the following recursive definitions
  - $V(i, W')$ = max value with items $S \subseteq \{1,\ldots,i\}$ s.t. $\text{weight}(S) \leq W'$
  - $C(i, V')$ = min weight with items $S \subseteq \{1,\ldots,i\}$ s.t. $\text{value}(S) \geq V'$
- Approx. alg. by rounding values $\tilde{v}_i = \lfloor v_i / Z \rfloor$ and running second alg.
- Is there an approx. alg. by rounding $\tilde{w}_i = \lfloor w_i / Z \rfloor, \tilde{W} = \lfloor W/Z \rfloor$ and running the first alg.?
  - Doing this will yield *some* subset $S \subseteq \{1,\ldots,n\}$
  - Trouble is that this new set may not be **feasible** for the original weight constraints

Knapsack overview

- **Input:** n items of integer values v_i and weights w_i and weight threshold W.
- **Input length:** $O(n \log VW)$
- **Output:** optimal $S \subseteq [n]$ maximizing $\text{value}(S)$ s.t. $\text{weight}(S) \leq W$
- **Various algorithms:**
    - Brute force alg: Runtime of $O(n2^n \log VW)$
    - DP alg: Runtime $O(nW \log VW)$ or $O(nV \log VW)$
    - ϵ-approx. alg: Runtime $O\left(\frac{n^3 \log VW}{\epsilon}\right)$

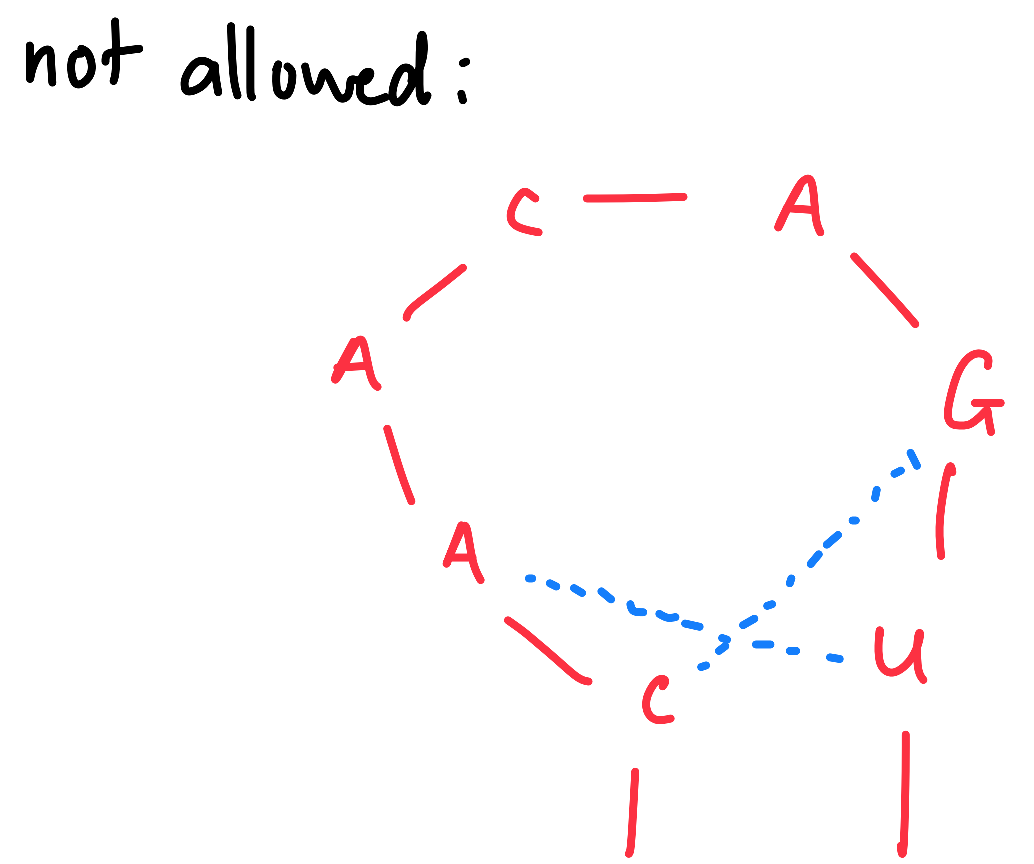
RNA secondary structure

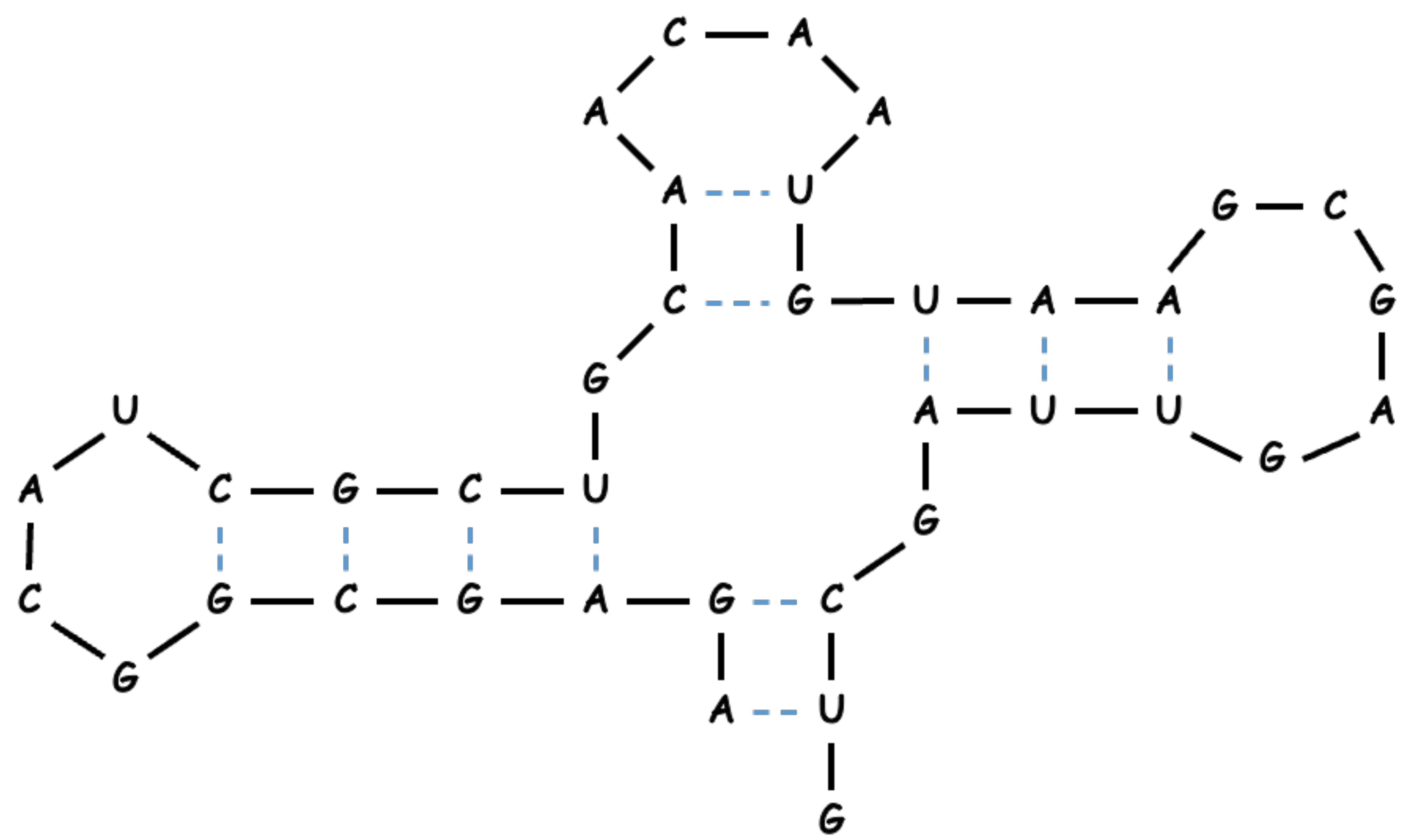
- RNA is expressed as a sequence of nucleotides: a string $B = b_1 \ldots b_n$ where each $b_i \in \{A, C, G, U\}$ for adenine, cytosine, guanine, and uracil.
- RNA tends to not be linear in a molecule and forms **secondary structures**
  - Secondary structures cause the molecule to loop back and forth
  - These are bonds between the base pairs



RNA secondary structure hypothesis

- **Definition.** A *secondary structure* for an RNA seq. $B = b_1 \ldots b_n$ is a set of pairs $S = \{(b_i, b_j)\}$ such that
  - WC condition: S is a matching and pairs are Watson-Crick complements i.e. $(b_i, b_j) \in WC := \{(A, U), (U, A), (G, C), (C, G)\}$
  - No sharp bends: $(b_i, b_j) \in S$ only if $4 < |i - j|$
  - Non-crossing: If (b_i, b_j) and (b_k, b_ℓ) then the intervals $[i, j]$ and $[k, \ell]$ are either disjoint or one contains the other.





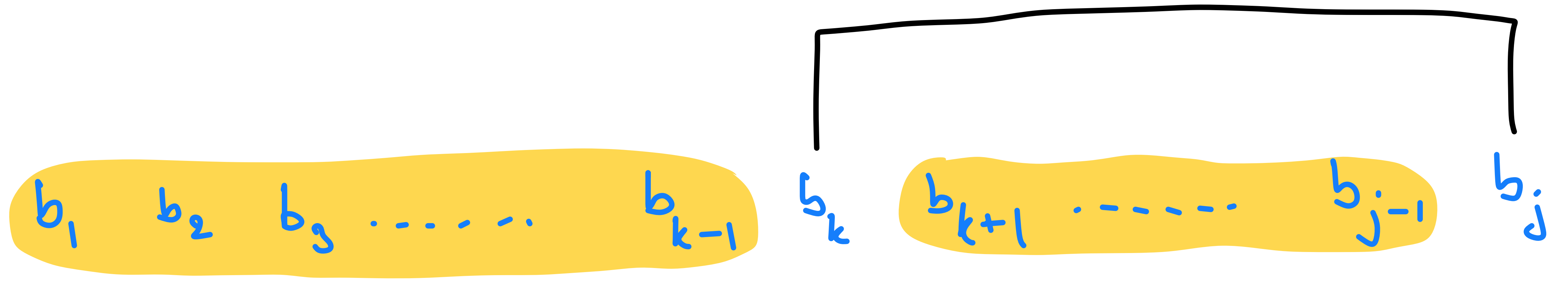
RNA secondary structure problem

- **Input:** an RNA seq. $B = b_1 \ldots b_n$
- **Output:** a secondary structure S of maximal size for B.

- **Dynamic programming attempt 1:** For $1 \le i \le j \le n$ define $S(j)$ as the maximal secondary structure using bases only $b_1, b_2, \ldots, b_j$. Let $f(j) = |S(j)|$.

RNA secondary structure problem

- **Two possibilities:** In the optimal solution, either $(b_k, b_j) \in S$ or $(b_k, b_j) \notin S$



- Splits problem into smaller problems but they aren't subproblems.
- **Problem:** Our choice of subproblem was not expressive enough.

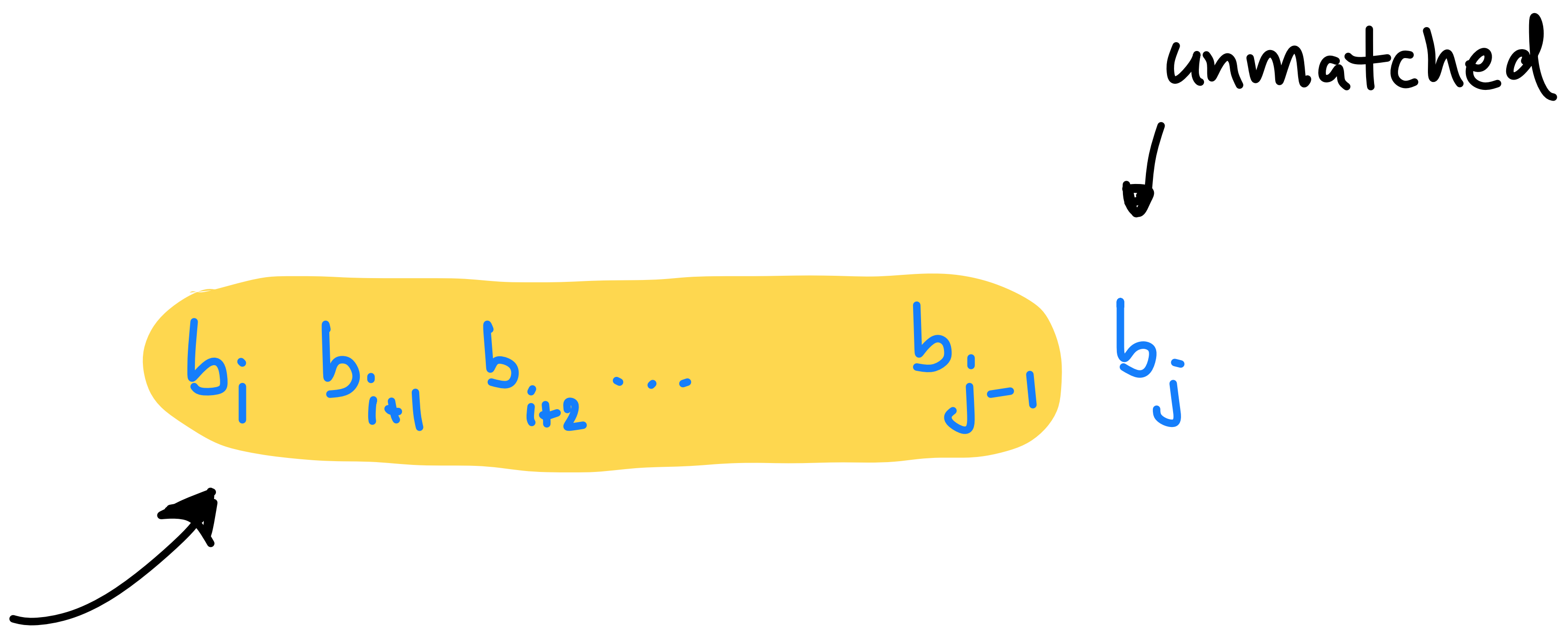
RNA secondary structure problem

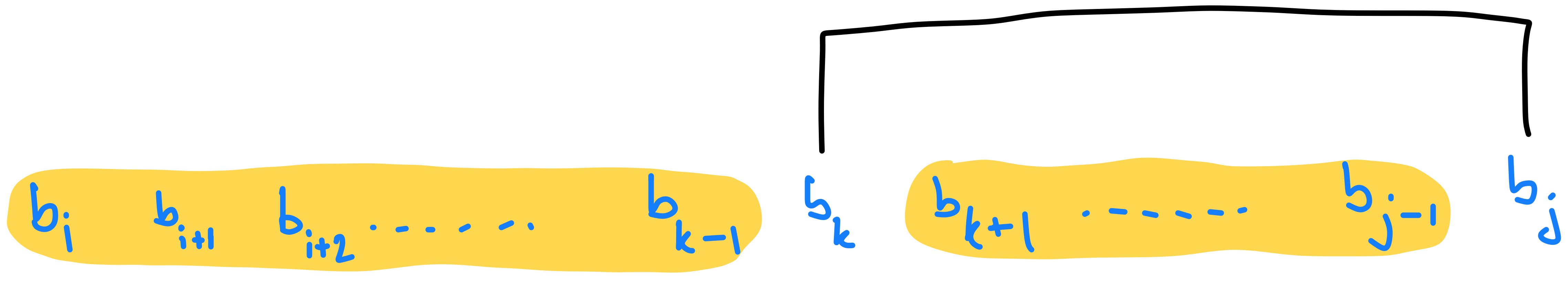
- **Input:** an RNA seq. $B = b_1 \ldots b_n$
- **Output:** a secondary structure S of maximal size for B.

- **Dynamic programming intuition:** For $1 \leq i \leq j \leq n$ define $S(i,j)$ as the maximal secondary structure using bases only $b_i, b_{i+1}, \ldots, b_j$. Let $f(i,j) = |S(i,j)|$.

RNA secondary structure DP algorithm

- **Dynamic programming intuition:** For $1 \leq i \leq j \leq n$ define $S(i, j)$ as the maximal secondary structure using bases only $b_i, b_{i+1}, \ldots, b_j$. Let $f(i, j) = |S(i, j)|$.
- **Recursive definition**:
  - In optimal solution, either b_j is not in a SS or (b_k, b_j) is in the SS
  - In first case, $f(i, j) = f(i, j-1)$ and $S(i, j) = S(i, j-1)$
  - In second case, $f(i, j) = 1 + f(i, k-1) + f(k+1, j-1)$
  - Optimal solution can be calculated as a recursive minimization





RNA secondary structure DP algorithm

- **Recursive definition**:
  - In optimal solution, either b_j is not in a SS or (b_k, b_j) is in the SS
  - In first case, $f(i,j) = f(i,j-1)$ and $S(i,j) = S(i,j-1)$
  - In second case, $f(i,j) = 1 + f(i,k-1) + f(k+1,j-1)$
- **Observation:** The recursive definition of $f(i,j)$ only depends on $f(i',j')$ for $|j' - i'| < |j - i|$.
  - Therefore, we fill memo from bottom-to-top w.r.t $|j - i|$.

RNA secondary structure DP algorithm

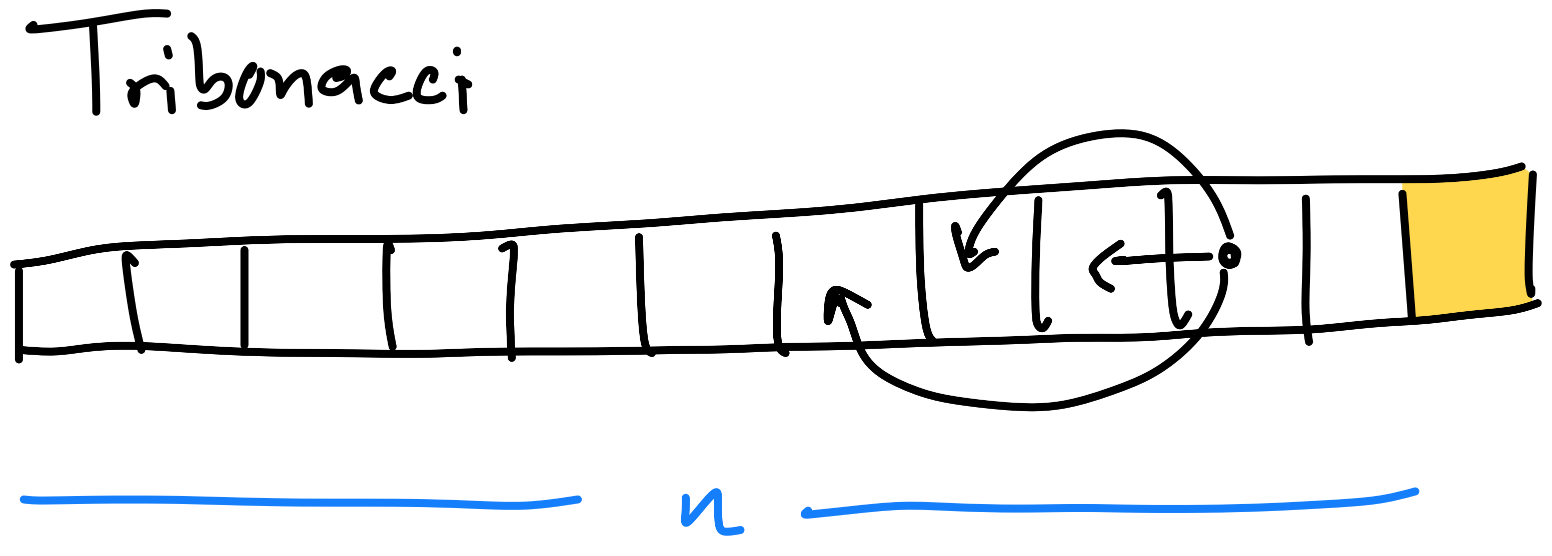
- **Filling memoization tables:**
  - Construct $n \times n$ tables M and f initialized as $\perp$
  - Set $f(i, i) \leftarrow 0$ for all i.
  - For $z \leftarrow 0$ to $n - 1$ and $i \leftarrow 1$ to $n - z$ ← iterate over length of interval z
    - Let $j \leftarrow i + z$
    - Compute $V \leftarrow \max_{k \in \{i,\ldots,j-5\} \wedge (b_j, b_k) \in WC} 1 + f(i, k - 1) + f(k + 1, j - 1)$ and let k be its argmin. ← valid partner k w.r.t. WC and sharp corner conditions
    - If $V > f(i, j - 1)$, set $f(i, j) \leftarrow V$ and set $M(i, j) \leftarrow k$
    - Else, set $f(i, j) \leftarrow f(i, j - 1)$ and keep $M(i, j) = \perp$.

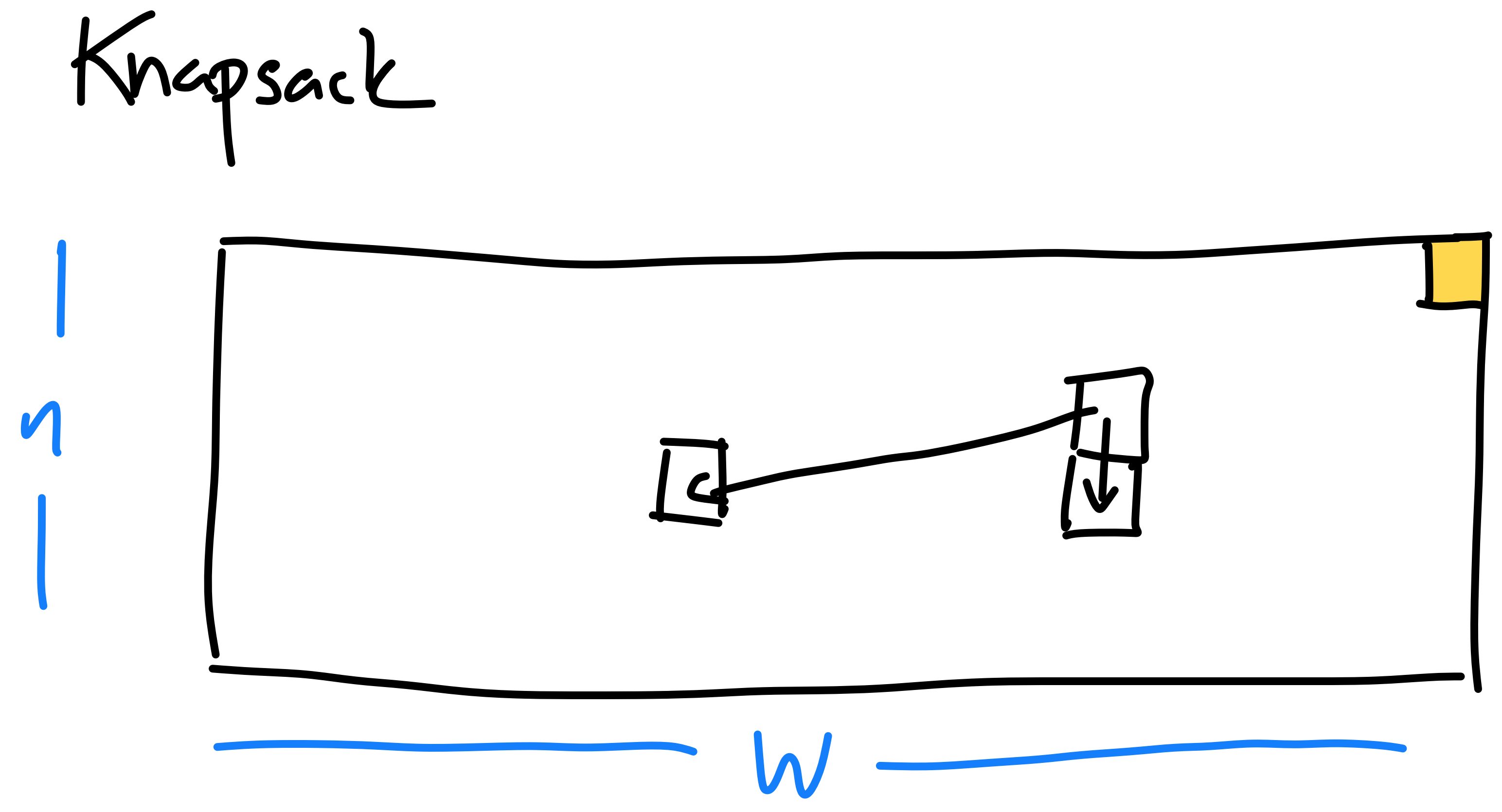
← record secondary structure in optimal sol.

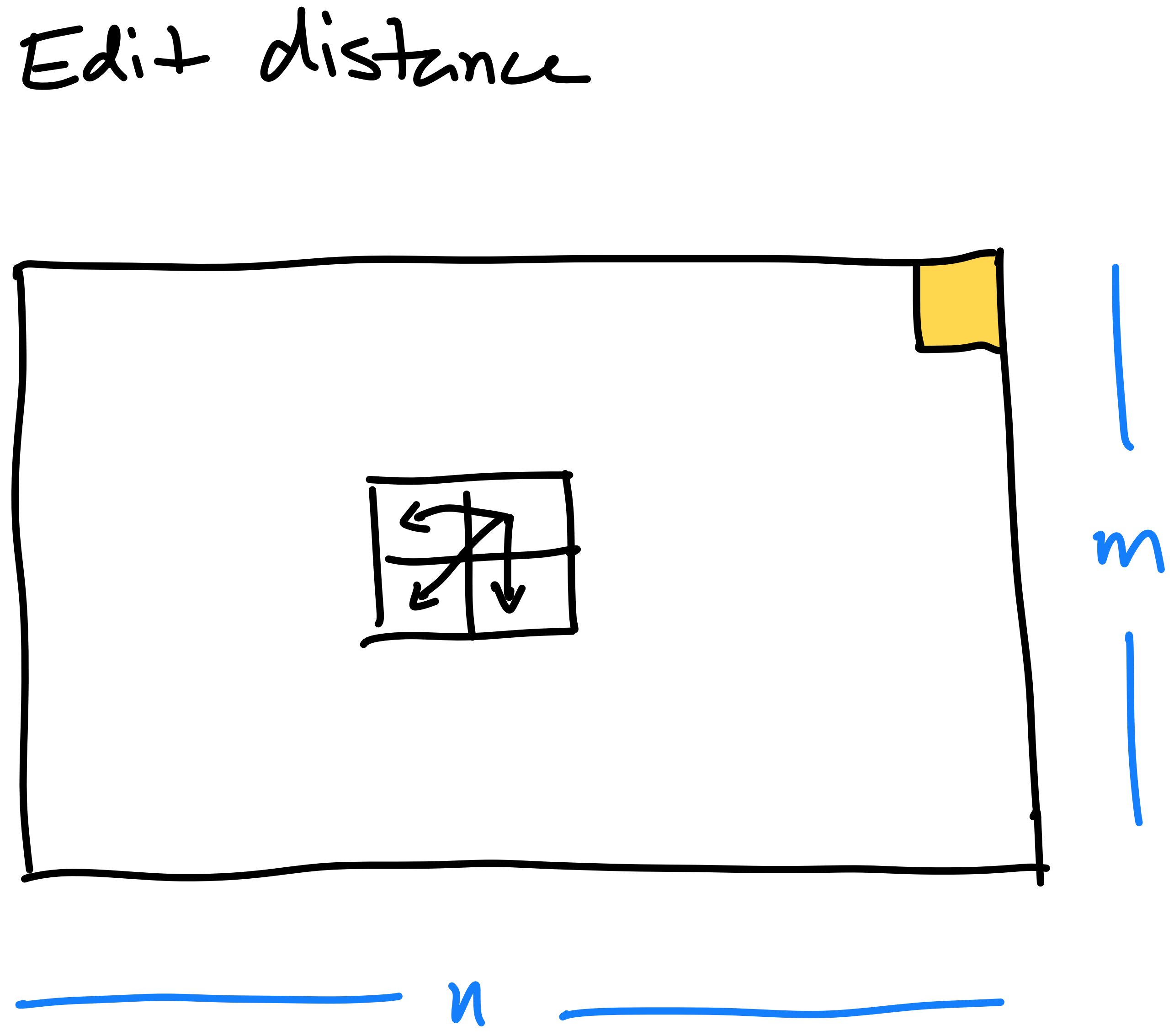
RNA secondary structure DP algorithm

- **Computing optimal secondary structure:**
- If $M(i, j) = k$ this means that $(b_k, b_j) \in S$. Else j is not included in S.
- To calculate optimal secondary structure run $\text{Print}(1,n)$ where
- $\text{Print}(i, j)$**:**
  - If $M(i, j) \leftarrow k$ output $(k, j) \cup \text{Print}(i, k-1) \cup \text{Print}(k+1, j-1)$
  - Else, output $\text{Print}(i, j-1)$
- Can be made to run faster in practice using DFS or BFS instead of recursion
- **Runtime:** $O(n^2)$ sized table with each recursive computation taking $O(n)$ time. Print runs in $O(n)$ time after the table is computed. Total runtime: $O(n^3)$.

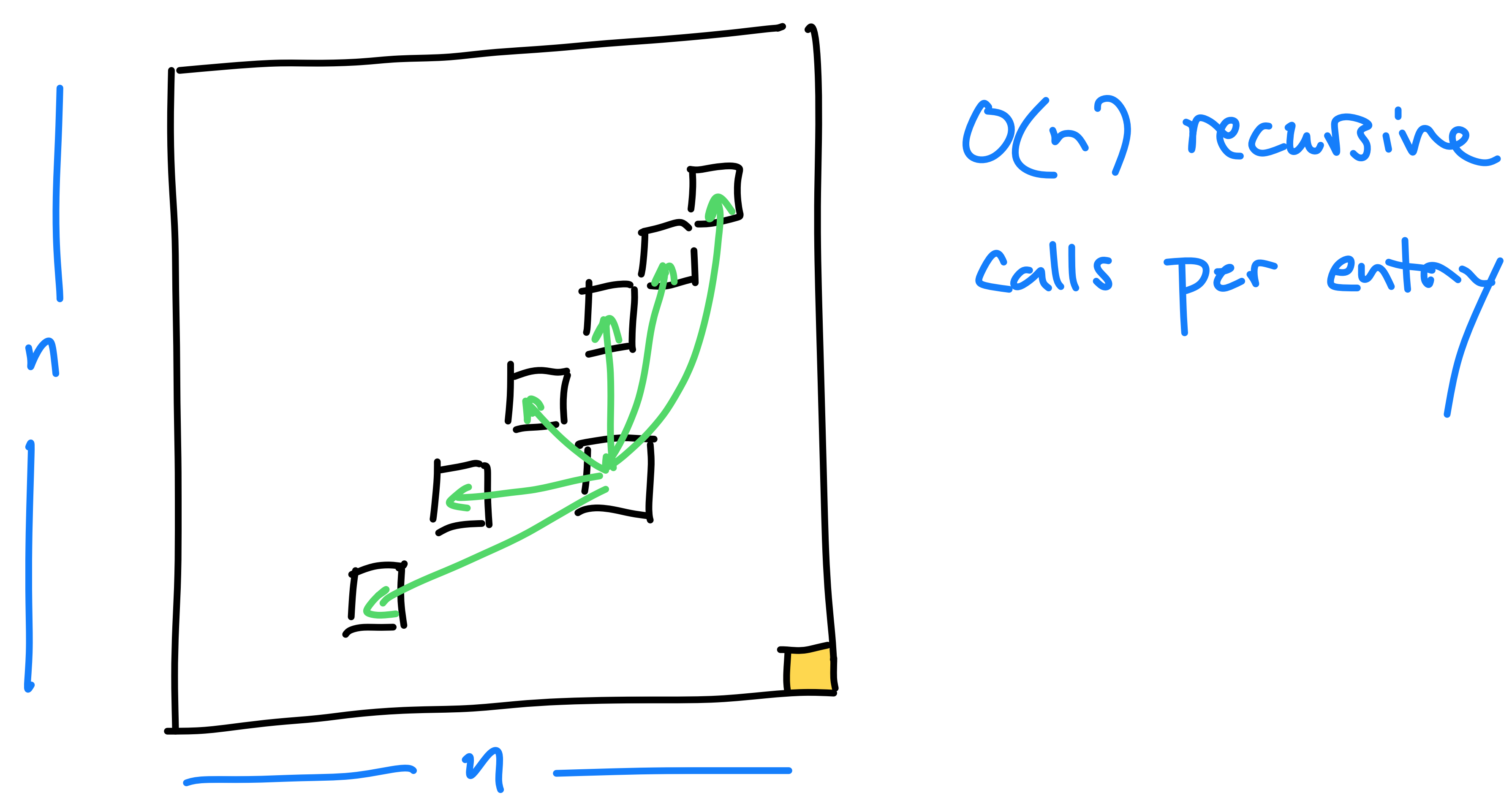
Dynamic programming patterns







RNA second sequence



Top-down vs bottom-up DP algorithms

- So far we have seen that the recursive subproblems in DP algorithms are always smaller. Examples
  - Knapsack: $f(n, W')$ depends on $f(n-1, W'')$ for $W'' \leq W'$
  - RNA SS: $f(i, j)$ depends on $f(i', j')$ where $|j' - i'| < |j - i|$
- Yields a "bottom-up" ordering for filling the memoization table
- Instead we could fill up the table "top-down"

Top-down vs bottom-up DP algorithms

- In a "top-down" DP algorithm $f(x)$
    - Conclude that $f(x)$ can be defined recursively based on $f(y_1), f(y_2), \ldots f(y_k)$
    - For each y_j, check if $f(y_j)$ has been previously calculated
        - If yes, use the value of $f(y_j)$
        - If not, recursive compute $f(y_j)$
- Overall, runtime is asymptotically the same! Each square of the memo is only computed once.

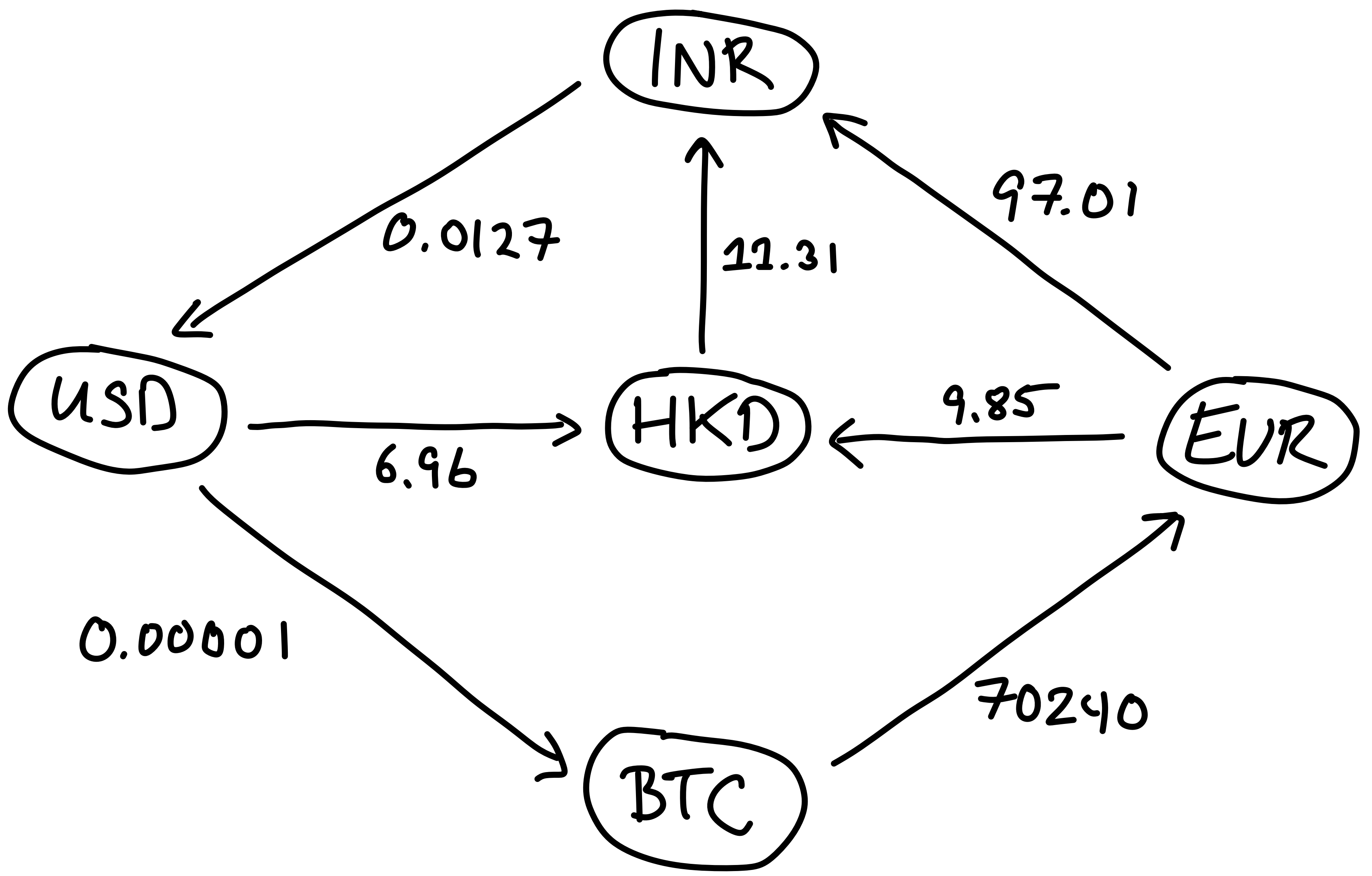
Top-down vs bottom-up DP tradeoffs

- In top-down approaches, not all squares may get calculated
    - Can yield constant factor savings in terms of runtime
- However, the recursion stack usually scales poorly in top-down approaches
    - For example, in Tribonacci, recursion stack would be $\Omega(n)$ in depth
    - Recursion stack is often in computer's memory while data being manipulated is expressed on the hard drive
    - Can yield memory overflow errors if not carefully programmed
- Top-down is better when the order of filling out squares isn't well defined
    - Occurs in graph DP algorithms like Bellman-Ford which we see soon
    - In such cases, a more sophisticated analysis is needed to argue that recursive defs. are not cyclical

Graph dynamic programming

Currency exchange

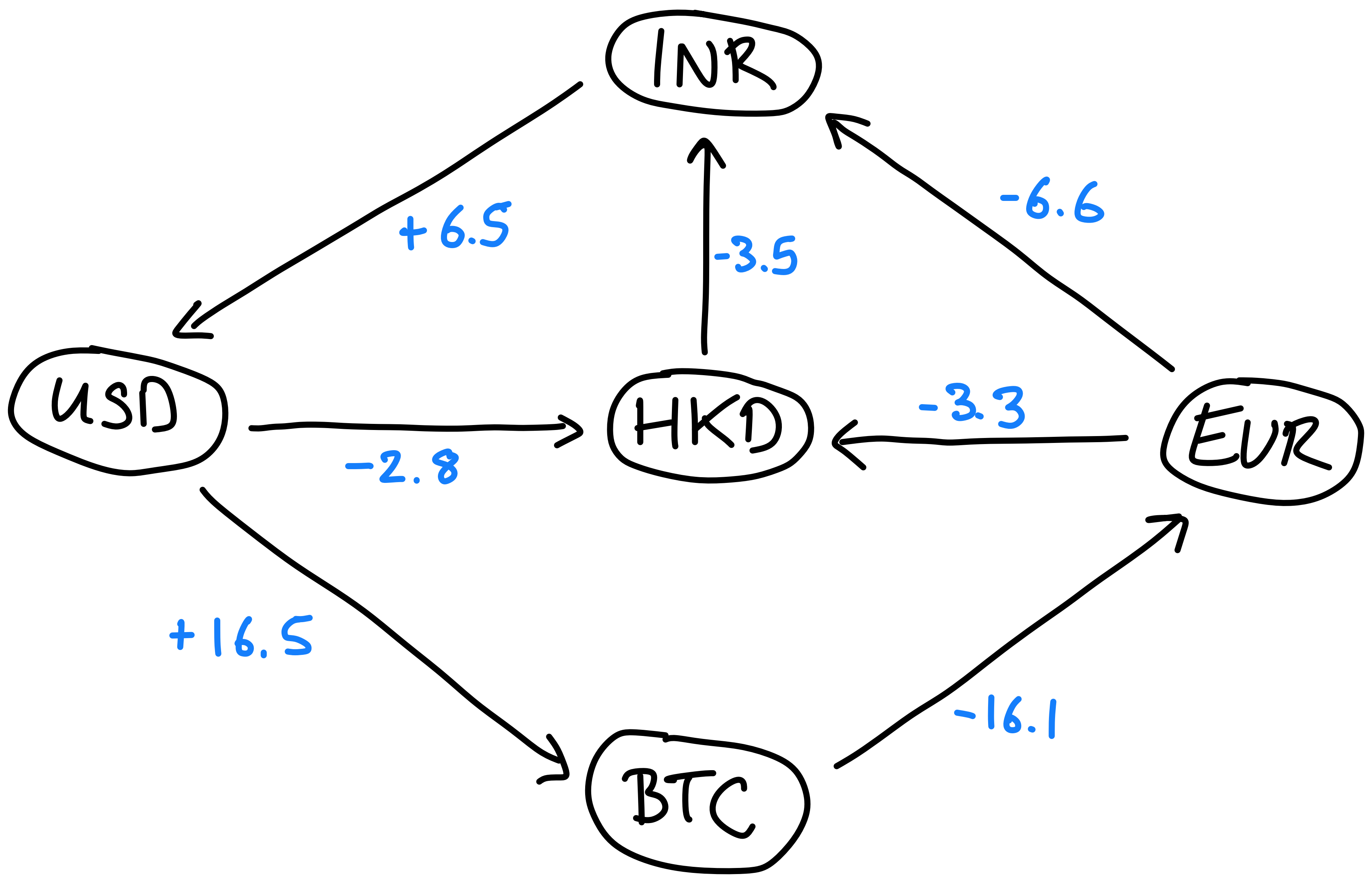
- USD to BTC: 0.00001
- BTC to EUR: 70,240
- INR to USD: 0.0127
- EUR to INR: 97.01
- EUR to HKD: 9.85
- HKD to INR: 11.31
- USD to HKD: 6.96



Currency exchange

- USD to BTC: 0.00001
- BTC to EUR: 70,240
- INR to USD: 0.0127
- EUR to INR: 97.01
- EUR to HKD: 9.85
- HKD to INR: 11.31
- USD to HKD: 6.96

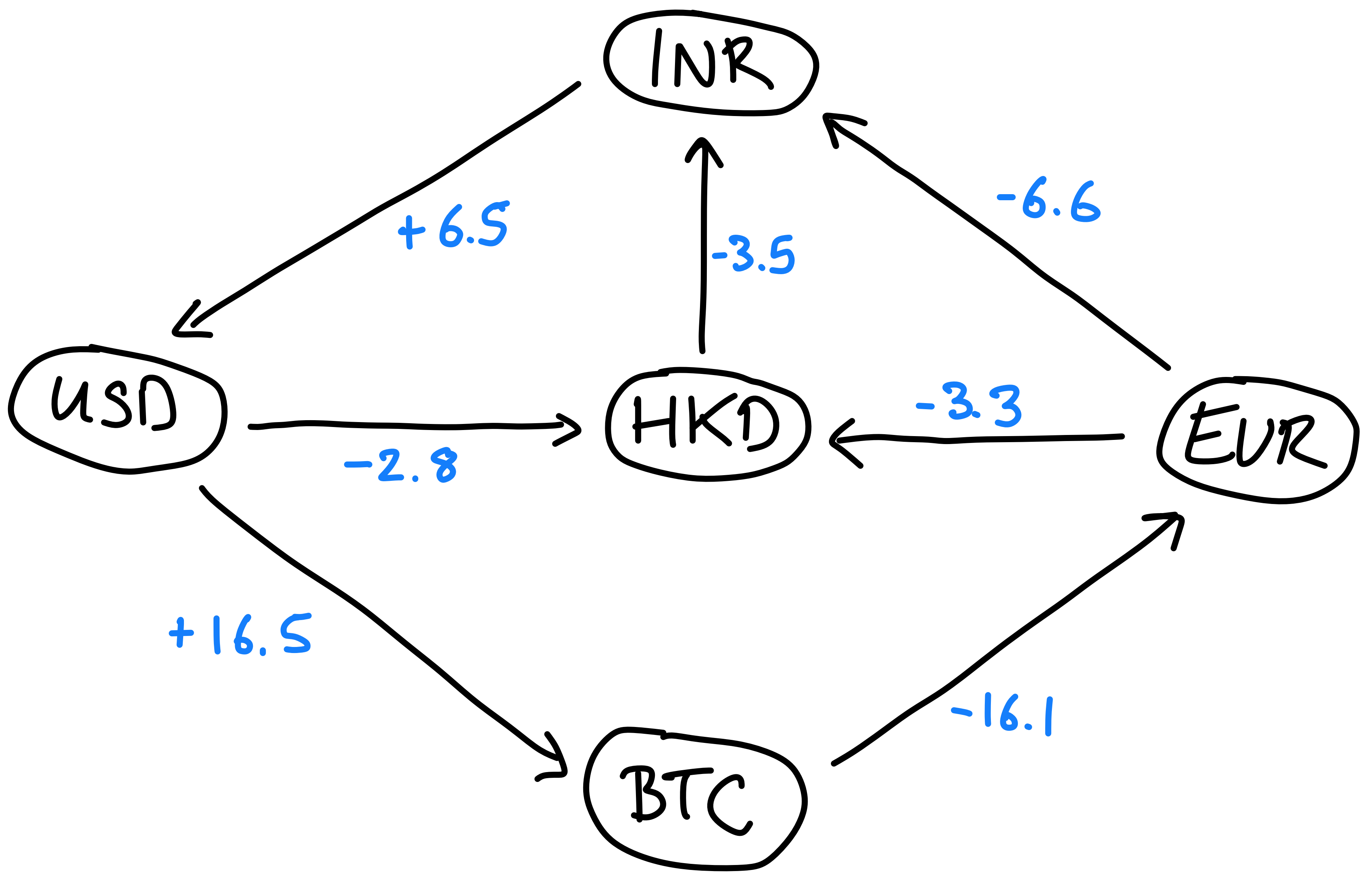
Set edge weight to $\log_2(1/r) = -\log_2(r)$



Currency exchange

Set edge weight to $\log_2(1/r) = -\log_2(r)$

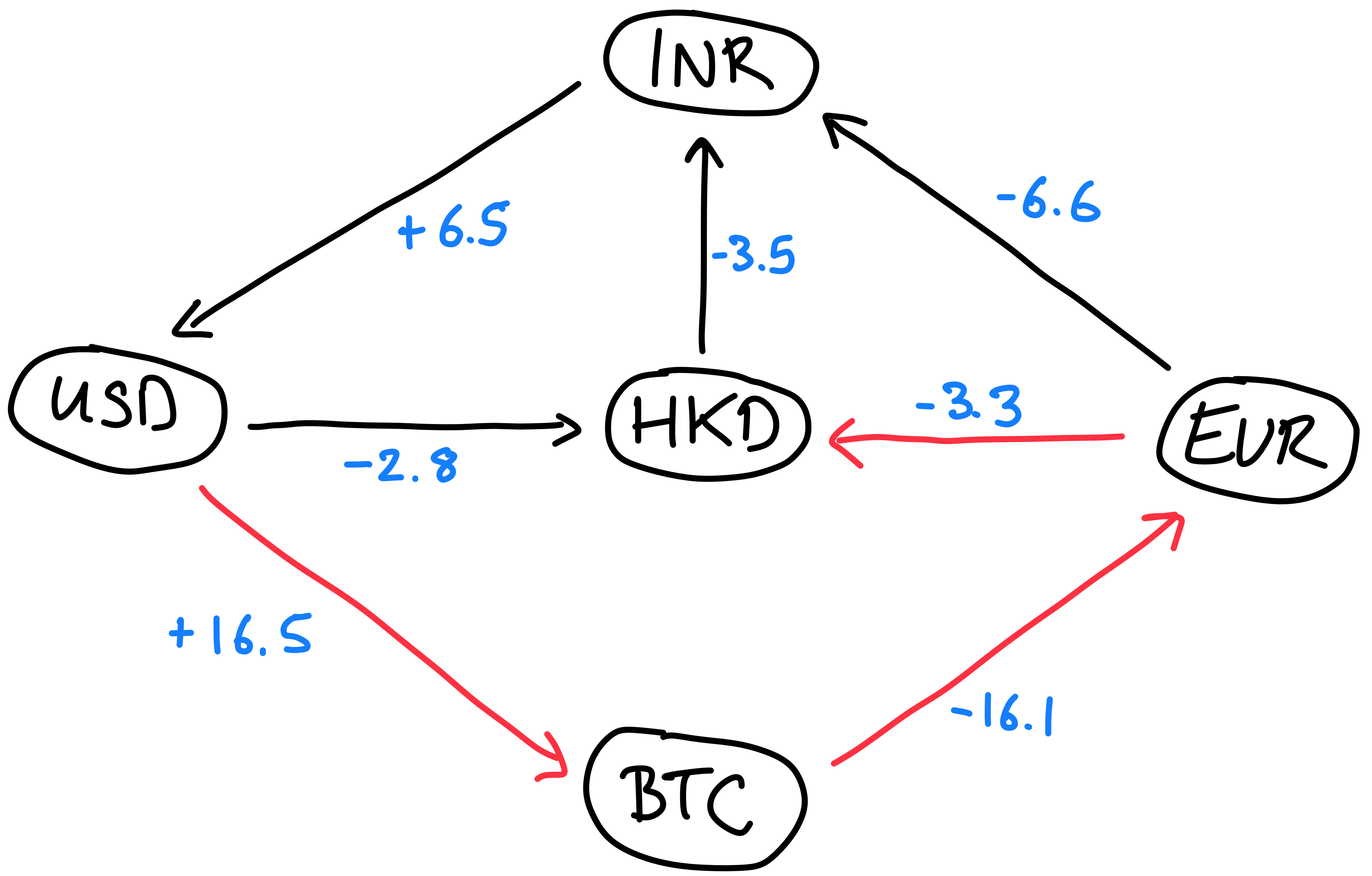
- A path $p : u \rightsquigarrow v$ of net weight w implies a currency conversion from 1 unit of u to 2^{-w} units of v
- Finding a path of least weight from u to v yields the best seq. of currency exchanges
- Direct conversion of USD to HKD yields $2^{2.8}$ HKD per USD



Currency exchange

Set edge weight to $\log_2(1/r) = -\log_2(r)$

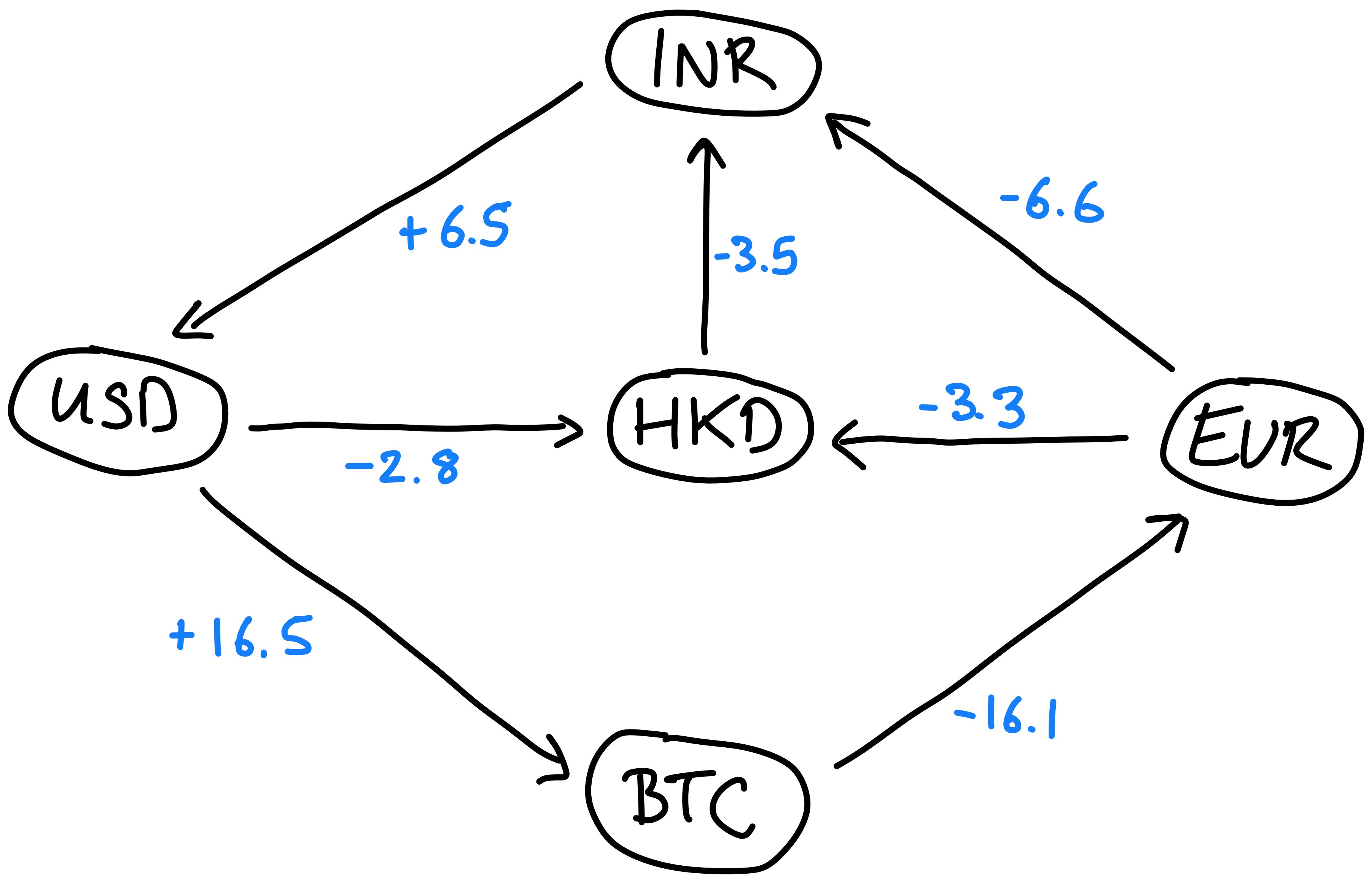
- A path $p : u \rightsquigarrow v$ of net weight w implies a currency conversion from 1 unit of u to 2^{-w} units of v
- Finding a path of least weight from u to v yields the best seq. of currency exchanges
- Direct conversion of USD to HKD yields $2^{2.8}$ HKD per USD
- USD→BTC→EUR→HKD yields $2^{-(16.5-16.1-3.3)} = 2^{2.9}$ HKD per USD



Currency exchange

Set edge weight to $\log_2(1/r) = -\log_2(r)$

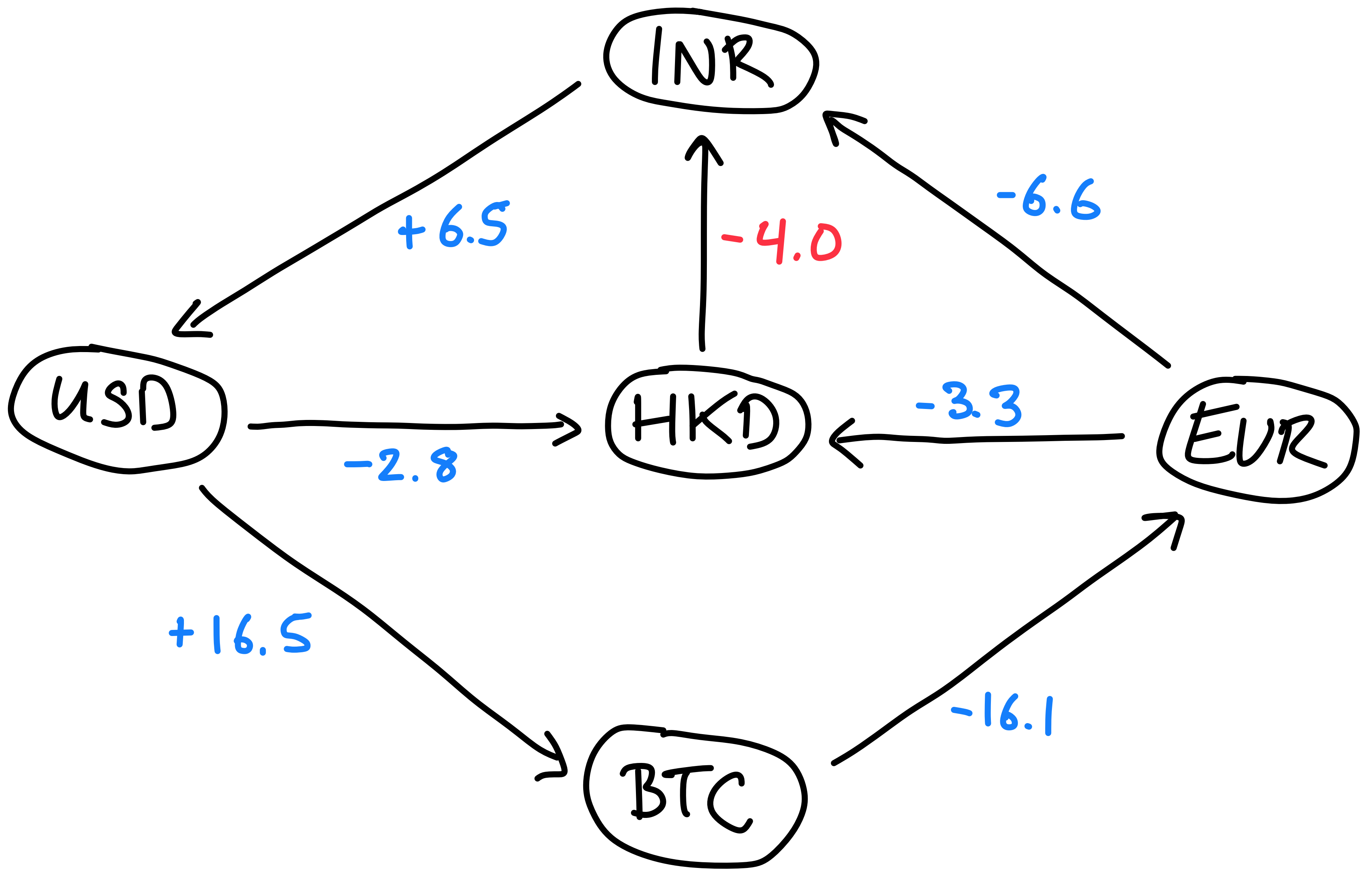
- What happens if HKD to INR rate changes from $2^{3.5}$ to $2^{4.0}$?



Currency exchange

Set edge weight to $\log_2(1/r) = -\log_2(r)$

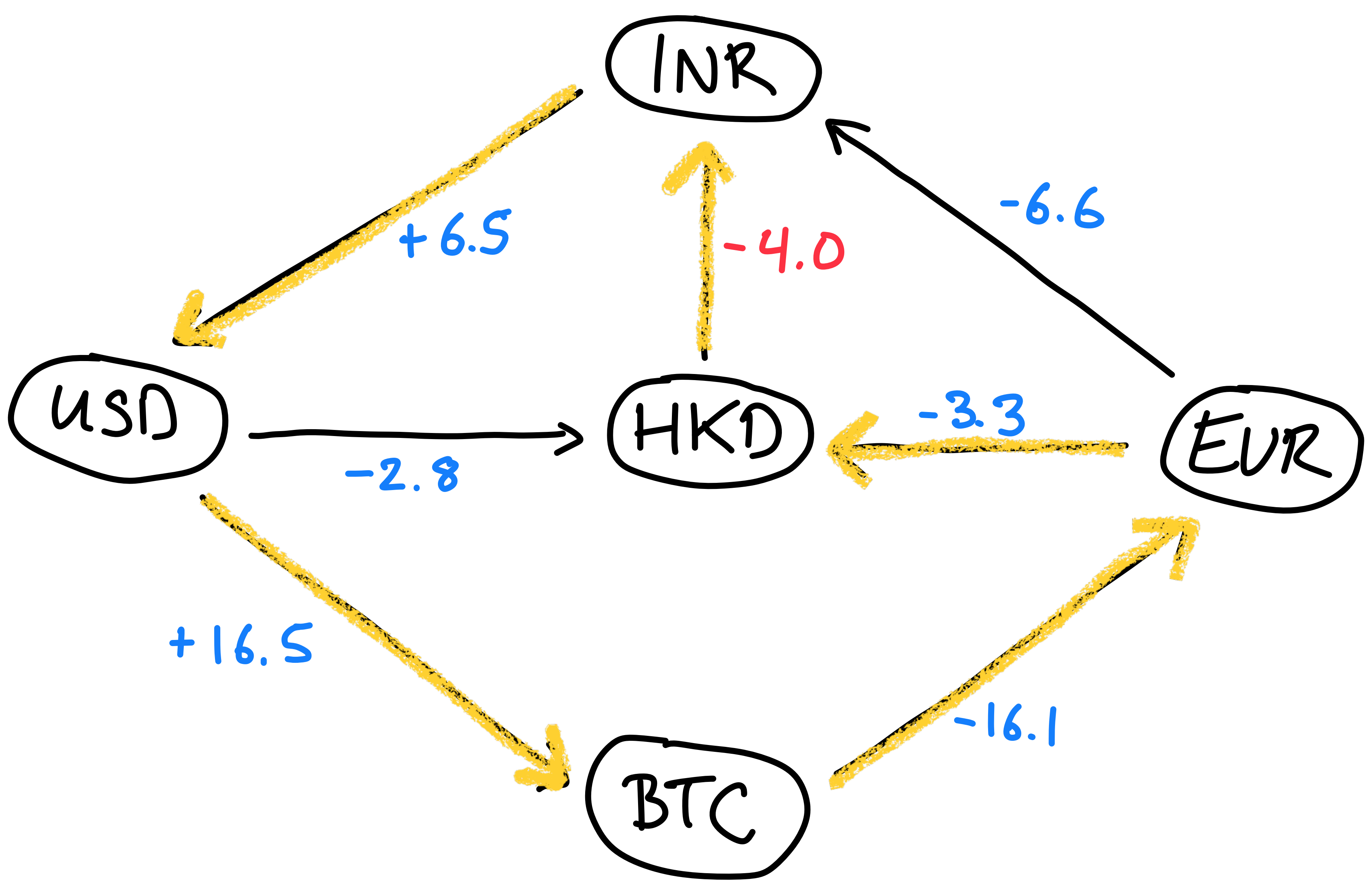
- What happens if HKD to INR rate changes from $2^{3.5}$ to $2^{4.0}$?



Currency exchange

Set edge weight to $\log_2(1/r) = -\log_2(r)$

- Consider the highlighted path from USD to USD:
- Converts 1 USD to $2^{0.8} > 1$ USD
- Constitutes a **negative cycle** in the graph
- In the currency exchange problem, negative cycles represent **arbitrage**
- Since there is a negative cycle, any currency can be converted into any other for arbitrarily cheap as the graph is strongly connected



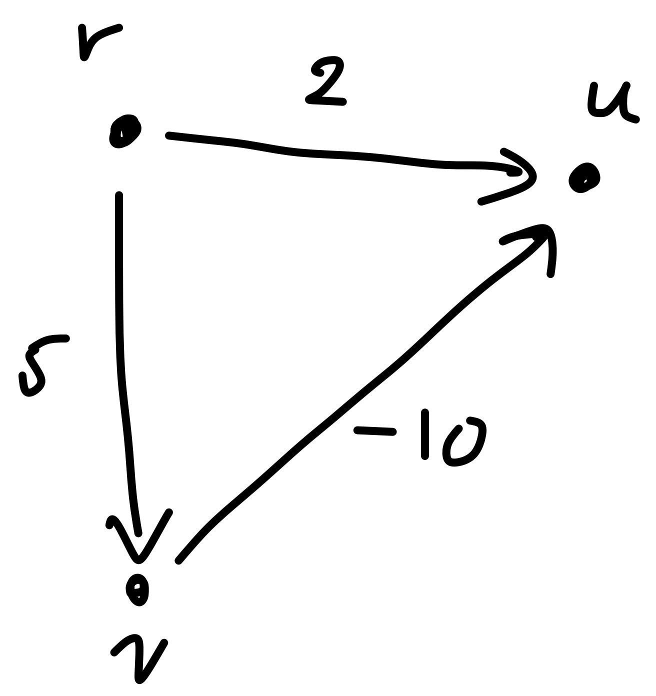
Negative weights shortest paths

- **Input:** A directed graph $G = (V, E)$ with weights $w : E \to \mathbb{R}$ and a vertex r
- **Output:** For every vertex v, the distance of the **lightest** directed path $r \rightsquigarrow v$ where a path's weight is the sum of its weights

- Why not just run Dijkstra's?
- Dijkstra's will incorrectly calculate distances when negative weights are involved

Negative weights shortest paths

- **Dijkstra's property:** Once a vertex v is visited, the distance $d(r, v)$ never needs updating again
    - This does not hold with negative weights
    - Need a slower but more careful algorithm that accounts for negative weights
- In this example,
    - Dijkstra's would set distance of u as 2 with path $r \to v$ in its first step
    - However, need to update the distance of u to -5 after v is visited.



Negative weights shortest paths

Applications

- Trade routes: each vertex is a commodity and edge $x \to y$ of weight w means 1 unit of x can be exchanged for 2^{-w} units of y
  - Multiplicative gains can be converted to linear gains by taking logarithms
  - Negative weights imply multiplicative losses
- Chemical networks: cost represent the excess energy required or **released** when a transformation is made
- Subsidies offered by governments for certain trades being performed
  - Example, US Govt. subsides flights from Portland, Oreg. to Pendleton, Oreg. to incentive airlines to fly to this market. (Annually, about $4 million for just this route)
  - How can an airline design its route network to maximize revenue in light of subsidies?

The Bellman-Ford algorithm

- Dijkstra's is a **greedy** algorithm and suffices to calculate shortest/lightest paths when all weights are non-negative
  - Distances will never need to be recalculated once set
- Bellman-Ford is a **dynamic programming** algorithm for computing shortest path in directed graphs
  - Will run slower than Dijkstra's: $O(mn)$ time versus $O(n + m)\log n)$ time
  - Will involve "resetting" distances as the algorithm goes along
  - Bellman-Ford will detect **negative cycles** as shortest paths are undefined if there are negative cycles