Algorithm Design and Analysis

Dynamic Programming (Again)

Roadmap for today

- More ***dynamic programming***

Key element #1: Memoization

Memoization: Don’t solve the same problem twice

```
function fib(int n) {
  if n <= 1 then  return 1
  else return fib(n-1) + fib(n-2)
}
```

```
dictionary<int,int> memo

function fib(int n) {
  if n <= 1 then  return 1
  if n is in not in memo then {
    memo[n] = fib(i-1) + fib(i-2)
  }
  return  memo[n]
}
```

Key element #2: Optimal substructure

Optimal substructure: Break the problem into smaller versions of itself (recursively), and build the solution to the bigger problem by combining the answers to the smaller (sub-)problems

Examples:

- *LCS(i,j)* = Length of the LCS between $S[\dots i]$ *and* $T[\dots j]$
- *V(k,B)* = Maximum value subset of items $1 \dots k$ with total weight $\leq B$
- *W(v)* = Max-weight independent set of the subtree rooted at v

“Recipe” for dynamic programming

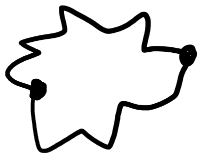
1. ***Identify a set of optimal subproblems***
   - Write down a clear and unambiguous definition of the subproblems.
2. ***Identify the relationship between the subproblems***
   - Write down a recurrence that gives the solution to a problem in terms of its subproblems
3. ***Analyze the required runtime***
   - *Usually* (but not always) the number of subproblems multiplied by the time taken to solve a subproblem.
4. ***Select a data structure to store subproblems***
   - *Usually* just an array. Occasionally something more complex.
5. ***Choose between bottom-up or top-down implementation***
6. ***Write the code!***

Mostly focus on these steps

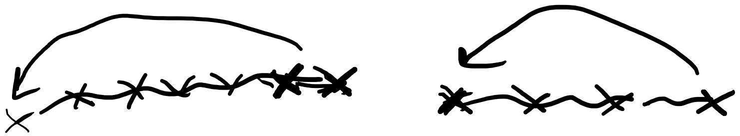
Problems!

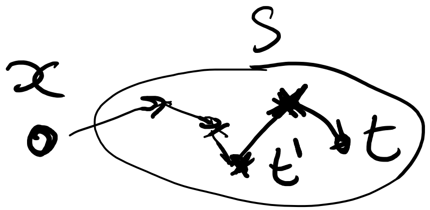
Traveling Salesperson Problem (TSP)

Definition (TSP) Given a complete, directed, weighted graph, we want to find a minimum-weight cycle that visits every vertex.



Start vertex x





subset of vertices

final vertex on path

$C(S, t)$ = shortest cost path $x \to t$ touches all S

Traveling Salesperson Problem (TSP)

$$C(S,t) = \begin{cases} w(x,t) & \text{if } S = \{x,t\} \\ \displaystyle\min_{\substack{t' \in S \\ t' \notin \{x,t\}}} C(S-\{t\}, t') + w(t',t) & \end{cases}$$

$$\text{answer} = \min_{t \in V - \{x\}} C(V,t) + w(t,x)$$

$$x \rightsquigarrow t' \rightarrow t$$

Traveling Salesperson Problem (TSP)

Analysis: TSP can be solved in $O(n^2 2^n)$ time

$2^n \cdot n$ subproblems

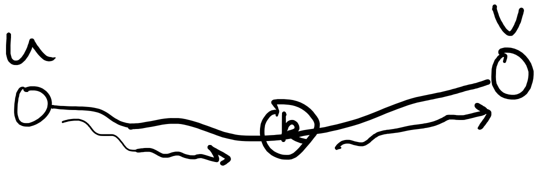
n time per

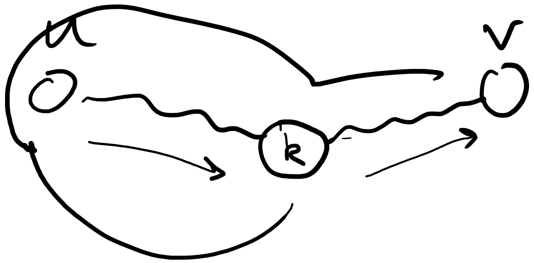
Data structure: How do we store the subproblems??

Use a bitset

All-pairs shortest paths

Definition (APSP) Given a directed, weighted graph, compute the length of the shortest path between every pair of vertices.





$D(u, v, k)$ = shortest path $u \to v$ using $1 \ldots k$

All-pairs shortest paths

$$D(u,v,\underline{\underline{k}}) = \min\big(D(u,v,k-1),\ D(u,k,k-1) + D(k,v,k-1)\big)$$

$$D(u,v,0) = \begin{cases} 0 & u = v \\ w(u,v) & (u,v) \in E \\ \infty & (u,v) \notin E \end{cases}$$

All-pairs shortest paths

Analysis: APSP uses $O(n^3)$ time and $O(n^3)$ <u>space</u>

n^3 subproblems, $O(1)$ time

That's a **lot** of space. Can we improve this?

All-pairs shortest paths

Optimization:

D[u][v] = base case from earlier → $D(u, v) = \begin{cases} 0, & \text{if } u = v \\ w(u, v), & \text{if } (u, v) \in E \\ \infty, & \text{otherwise} \end{cases}$

for k = 1 **to** n **do**
  for u = 1 **to** n **do**
    for v = 1 **to** n **do**
      $D(u,v) = \min\left(D(u,v),\ D(u,k) + D(k,v)\right)$

All-pairs shortest paths

Analysis: **Floyd-Warshall** runs in $O(n^3)$ time and $O(n^2)$ space

Why does it work?

$$u \xrightarrow[k?]{} k \xrightarrow[k?]{} v$$

Longest Increasing Subsequence

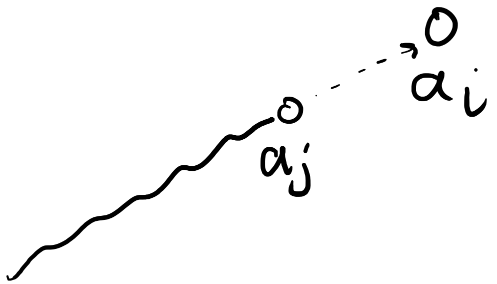
Definition (LIS) Given a sequence of numbers $a_1, a_2, \ldots, a_n$, find the length of the longest strictly increasing subsequence. i.e., find indices $i_1, i_2, \ldots, i_k$ such that $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$ for the largest possible k

$\times \quad \underline{2} \quad \times \quad \times \quad \underline{3} \quad \times \quad \times \quad \underline{5} \quad \times \quad \times \quad \underline{9} \quad \times \quad \times \quad 10 \quad \times \quad \times \quad \underline{12}$

$L(i)$ = LIS ending at a_i

Longest Increasing Subsequence

$$L(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 + \max\limits_{\substack{a_j < a_i \\ 0 \le j < i}} L(j) & \end{cases}$$



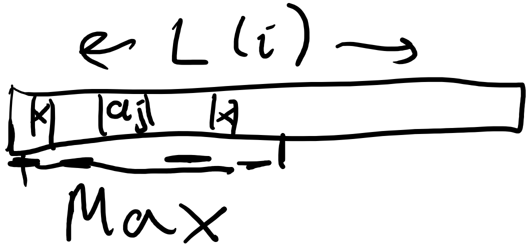
Longest Increasing Subsequence

Analysis: LIS can be solved in $O(n^2)$

n problems , n time $O(n^2)$

Faster? Seems a bit slow…

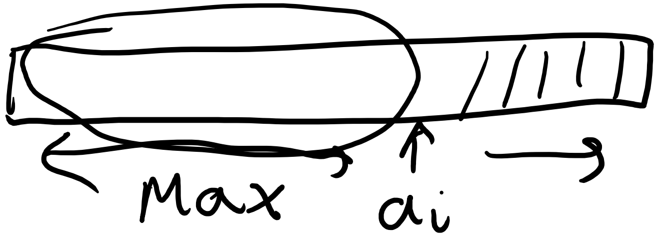
$$1 + \max_{\substack{0 \le j < i \\ \boxed{a_j < a_i}}} L(j)$$



Longest Increasing Subsequence

Optimizing with range queries

$$1 + \max_{\substack{0 \le j < i \\ a_j < a_i}} L(j)$$



n = size(a)
b = sorted(a)

LIS = **SegTree**(n+1, 0)

for i **in** 1 **to** n – 1 {

  rank = binary_search(b, a[i])

  LIS.Assign(rank, 1 + LIS.RangeMax(0, rank))

}

return LIS.RangeMax(0, n+1)

Take-home messages

- Breaking a problem into subproblems is hard. ***Common patterns:***
  - Can I use the first k elements of the input?
  - Can I restrict an integer parameter (e.g., knapsack size) to a smaller value?
  - On trees, can I solve the problem for each subtree? (Tree DP)

  - Can I store a subset of the input? (TSP subproblems)
  - Can I remember the most recent decision? (Previous vertex in TSP)

- Many techniques are useful to **optimize** a DP algorithm:
  - Can I remove redundant subproblems to save space? (Floyd-Warshall)
  - Can I use a fancier data structure than an array? (LIS with SegTree)