Dynamic Programming

Dynamic Programming: Overview

- Consider the problem of finding n-th Fibonacci Numbers:
  - Recursive Solution
    - F(n) = F(n-1) + F(n-2)
    - F(0) = 0
    - F(1) = 1
- Runtime Analysis
  - T(n) = T(n-1) + T(n-2) + O(1)
  - Exponential time (double exponential in term of input size)
    - Input size for this problem is **log n**
  - T(n) ≥ 2T(n-2)+ O(1) → n/2 levels in the recursion tree → $\Omega(2^{n/2})$
  - $T(n) \leq 2T(n-1) + O(1)$ → n levels in the recursion tree → $O(2^n)$

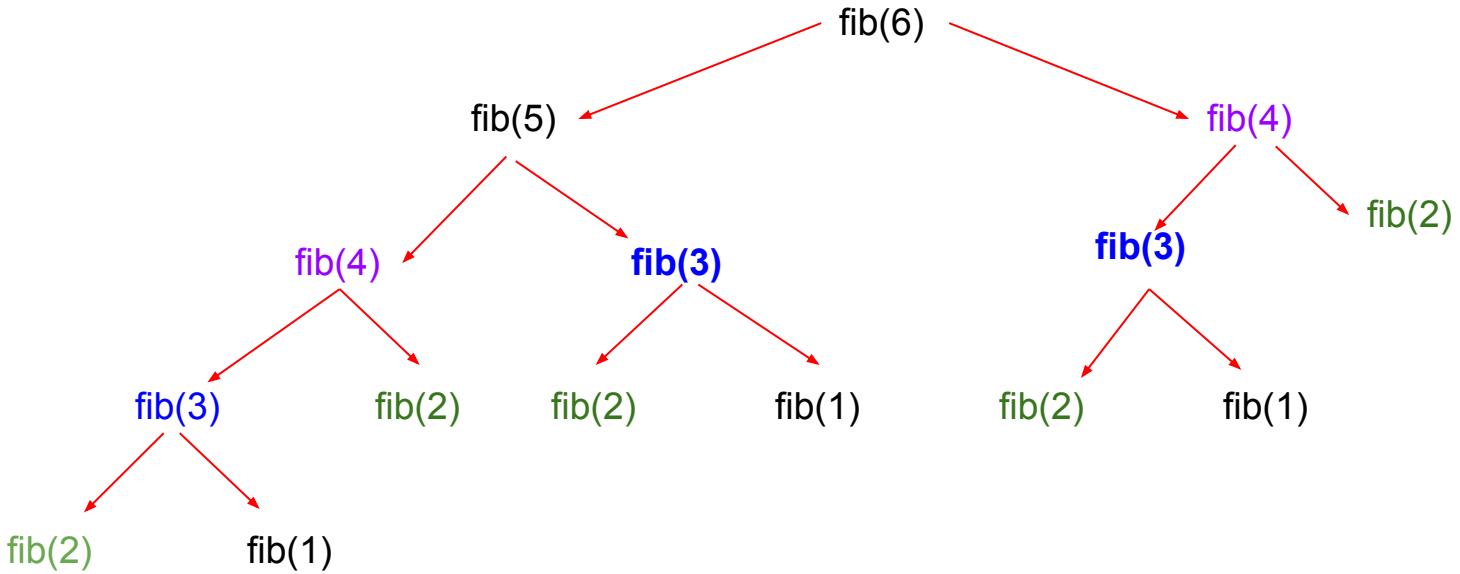
Recursive algorithm

```
def Fib(n):
        if n == 0, return 0
        if n == 1, return 1
        return Fib(n-1) + Fib(n-2)
```

Dynamic Programming: Overview

Why the recursive fibonacci is so slow?

Overlapping subproblems



Dynamic Programming: Overview

- An efficient implementation of fibonacci
- Runtime: O(n)
  - N-1 integer addition
  - Recursion tree is collapsed

Iterative algorithm

```
def betterFib(n):
    F[0] = 0
    F[1] = 1
    for i = 2, …, n:
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

Main idea of dynamic programming:

- solve “subproblems” from smaller to larger (**bottom up**) storing solutions.
- Runtime:
  #subproblems x time to solve one subproblem

Dynamic Programming: Overview

- Another efficient implementation of fibonacci: **memoization**
  - It is the same recursive algorithm
- Runtime: O(n)
  - Comes from Memo
  - It records the computation that is already done and therefore many of the function calls are not executed
  - And therefore many of the subproblems are not called again

```
F = [0,1,None, None, …, None]
def Fib(n):
    if F[n] != None:
        return F[n]
    else:
        F[n] = Fib(n-1) + Fib(n-2)
    return F[n]
```

When can we use dynamic programming?

- (Optimal) Recursive SubStructure
- Overlapping subproblems
- Polynomial number of distinct subproblems

When can we use dynamic programming?

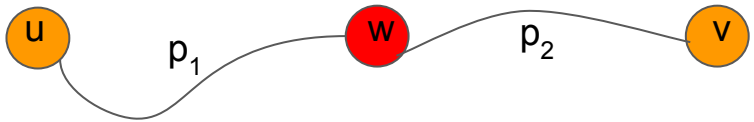
- **(Optimal) Recursive Substructure**
  - The solution to a problem can be expressed in terms of solutions to smaller sub-problems.
    - If it is not an optimization problem, determine if a solution can be expressed in terms of solutions to smaller subproblems
  - Example:
    - Big problems break up into sub-problems.
      - Fibonacci: F(i) for $i \leq n$
      - Fibonacci: $F(i) = F(i-1) + F(i-2)$

Optimal Substructure

- Optimal substructure does not always apply
  - **Shortest path problem**: Find a simple path from u to v consisting of the fewest edge.
    - Has an optimal substructure
  - **longest path**: find a simple path from u to v consisting of the most edges.
    - Simple path means a path with no cycle
    - Does not have an optimal substructure

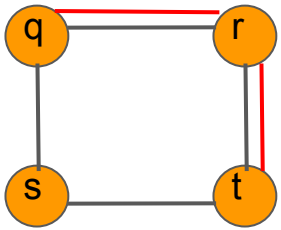
Optimal Substructure: shortest path problem

- **p**: the shortest path between u and v: optimal solution
- Number of edges in p = Number of edges in p_1 + Number of edges in p_2
- **Claim**: p_1 is a shortest path from u to w:
  - If there were another path, say p'_1 from u to w with fewer edges then we could cut out p_1 and paste in p'_1 to produce a path $p' = p'_1 + p_2$ with fewer edges → contradiction: p_1 is an optimal solution or shortest path
  - Similarly we can show p_2 is the shortest path from w to v
- Therefore we can find a shortest path from u to v by considering all intermediate vertices w, finding a shortest path from u to w, finding a shortest path from w to v, choosing the w with overall shortest path

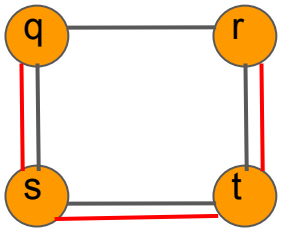


Optimal Substructure: longest path problem

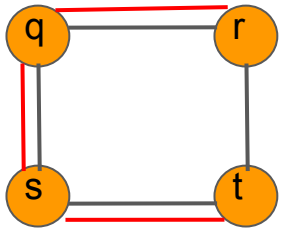
- Longest path from **q** and **t**: q-r-t
- Is q-r a longest path from q to r? No, longest path is q-s-t-r
- Is r-t a longest path from r to t? No, longest path is r-q-s-t
- This problem does not have optimal substrcuture→ we cannot use dynamic programming /greedy here



Longest path from q to t



Longest path from q to r



Longest path from r to t

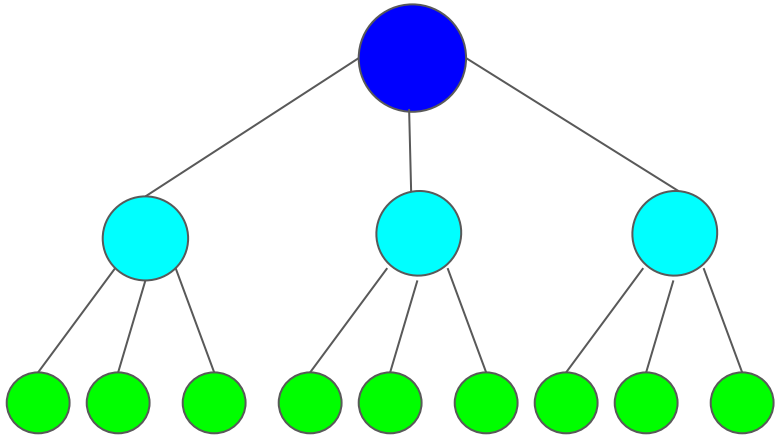
Overlapping Subproblems

- The subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems.
  - The sub-problems overlap
  - Fibonacci:
    - Both F[i+1] and F[i+2] directly use F[i].
  - And lots of different F[i+x] indirectly use F[i].

Implications: Save time (have more efficient algorithm) by solving a sub-problem just once and recording the result in a table from which a solution to the original problem can be obtained

Divide-and-conquer

Break up a problem into **independent** subproblems; solve each subproblem; combine solutions to subproblems to form solution to original problem.



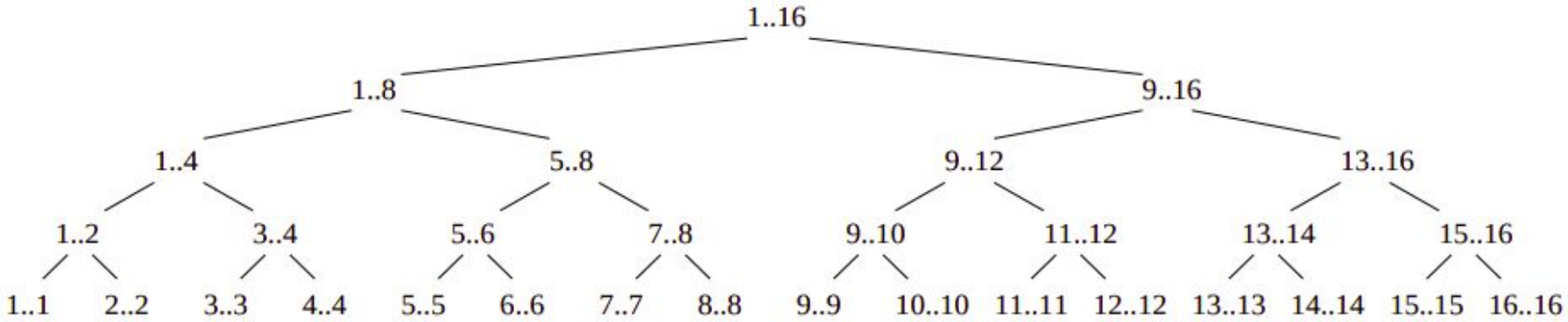
problem

sub-problem

sub-sub-problem

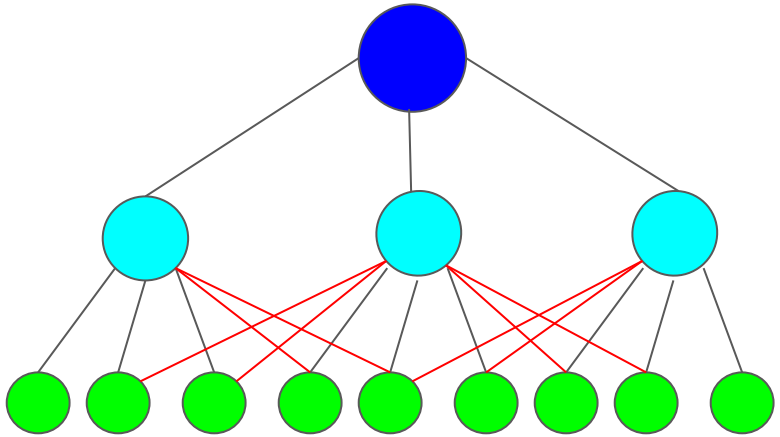
Subproblem graph view: mergesort

- No repeated subproblems
  - Dynamic Programming not good for “Divide & Conquer” algorithms



Dynamic programming

Break up a problem into a series of independent but **overlapping** subproblems; combine solutions to smaller subproblems to form solution to large subproblem.



problem

sub-problem

sub-sub-problem

Polynomial Subproblems

- The number of **distinct** subproblems is small enough to be evaluated in polynomial time.