- General framework
  - Break (a large chunk of) a problem into two smaller subproblems of the same type
  - Solve each subproblem recursively and independently
  - At the end, quickly combine solutions from the two subproblems and/or solve any remaining part of the original problem

- Theorem: Let $a \geq 1$ and $b > 1$ be constants, $f(n)$ be a function, and $T(n)$ be defined on non-negative integers by the recurrence $T(n) \leq a \cdot T\left(\frac{n}{b}\right) + f(n)$, where n/b can be $\left\lceil \frac{n}{b} \right\rceil$.
  Let $d = \log_b a$. Then:
  - If $f(n) = O\left(n^{d-\epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = O\left(n^d\right)$.
  - If $f(n) = O\left(n^d \log^k n\right)$ for some $k \geq 0$, then $T(n) = O\left(n^d \log^{k+1} n\right)$.
  - If $f(n) = O\left(n^{d+\epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = O\big(f(n)\big)$.

Greedy Algorithms

- Greedy/myopic algorithm outline
  - Goal: find a solution x maximizing/minimizing objective function f
  - Challenge: space of possible solutions x is too large
  - Insight: x is composed of several parts (e.g., x is a set or a sequence)
  - Approach: Instead of computing x directly…
    - Compute it one part at a time
    - Select the next part “greedily” to get the most immediate “benefit” (this needs to be defined carefully for each problem)
    - Polynomial running time is typically guaranteed
    - Need to prove that this will always return an optimal solution despite having no foresight

- What order?
  - Earliest start time: ascending order of s_j
  - Earliest finish time: ascending order of f_j
  - Shortest interval: ascending order of $f_j - s_j$
  - Fewest conflicts: ascending order of c_j, where c_j is the number of remaining jobs that conflict with j

Dynamic Programming

- Outline
  - Breaking the problem down into simpler subproblems, solve each subproblem just once, and store their solutions.
  - The next time the same subproblem occurs, instead of recomputing its solution, simply look up its previously computed solution.
  - Hopefully, we save a lot of computation at the expense of modest increase in storage space.
  - Also called "memoization"

- How is this different from divide & conquer?

Top-Down vs Bottom-Up

- Top-Down may be preferred…
  - …when not all sub-solutions need to be computed on some inputs
  - …because one does not need to think of the "right order" in which to compute sub-solutions
- Bottom-Up may be preferred…
  - …when all sub-solutions will anyway need to be computed
  - …because it is faster as it prevents recursive call overheads and unnecessary random memory accesses
  - …because sometimes we can free-up memory early

greedy

①什么时候 sort by start time / finish time

②问作业第四题还有 bonus