Asymptotic Analysis: T(n) is O(f(n)) if there exist constants c>0 and n0>=0 such that for all n>= n0 we have T(n)<=c ·f(n). If false set Counterexample.

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Greedy Algorithms: (a)An optimal algorithm is to schedule the jobs in decreasing order of w/t. We prove the optimality of this algorithm by an exchange argument. Consider any other schedule: As is standard in exchange argument, we observe that this schedule must contain an inversion. A pair of jobs i, j, for which i comes before j in the alternation solution, and j comes before i in the greedy algorithm. For this pair, we have w/tj ≥ w/ti, by the definition of the greedy schedule. If we can show that swapping this pair i, j does not increase the weighted sum of completion times, then we can iteratively do this until there are no more inversions, arriving at the greedy schedule without having increased the function we are trying to minimize → the greedy algorithm is optimal. (b)**Proof by contradiction:** Assume Greedy is different from OPT. Let's see what's different. Let i1 , i2 , …, ik denote the set of jobs selected by greedy. Let j1 , j2 , …, jm denote set of jobs in the optimal solution. Find largest possible value of r such that i1=j1,i2=j2,…,ir=jr. By definition of Greedy, job i r+1finishes before jr+1. Why not replace job jr+1with job ir+1. Solution still feasible and optimal, but contradicts maximality of r. **Cut lemma:** Let S be any subset of nodes, and let e be the min cost edge with one endpoint in S. Then every MST must contain e. **Prim:** 逐个加点. **Kruskal:** 逐条加边. Proof of Kruskal: 1.prove Kruskal produce a tree(acyclic and connected graph); 2.prove every edge chosen is in MST(Cut lemma); 3. Prove tree T is a MSL. **Reduced schedule claim:** any unreduced schedule can transform into reduced schedule with no more eviction.

Divide and Conquer: The important observation is that If A contains a majority element e, => e must be a maj. element in at least one of A[1..n/2] and A[n/2+1..n].Algorithm is then: Find majority e1, if it exists, in A[1..n/2], Find majority e2, if it exists, in A[n/2 + 1..n], If e1 exists, count how many times it occurs in A If more than n/2, report e1. If e2 exists, count how many times it occurs in A. If more than n/2, report e2.If neither exist, report, "no majority". The base case is when n=1, and we can simply return the only element as the majority. **Closest-pair:** T(n) = O(nlog2n)，可以优化至 O(nlogn). • Divide: draw vertical line L so that roughly ½n points on each side. • Conquer: find closest pair in each side recursively. • Combine: find closest pair with one point in each side Θ(n^2). • Return best of 3 solutions.

Dynamic Programming: **Weighted Interval Scheduling:**

```
Closest-Pair(p_1, …, p_n) {
   Compute separation line L such that half the points      O(n log n)
   are on one side and half on the other side.

   δ_1 = Closest-Pair(left half)                             2T(n / 2)
   δ_2 = Closest-Pair(right half)
   δ   = min(δ_1, δ_2)

   Delete all points further than δ from separation line L   O(n)

   Sort remaining points by y-coordinate.                    O(n log n)

   Scan points in y-order and compare distance between       O(n)
   each point and next 11 neighbors. If any of these
   distances is less than δ, update δ.

   return δ.
}
```

- **Notation.** OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, …, j
  - Case 1: OPT selects job j
    - Cannot use incompatible jobs {p(j) + 1, p(j) + 2, …, j-1}
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, …, p(j)
  - Case 2: OPT does not select job j
    - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, …, j-1

Optimal substructure

$$OPT(j)=\begin{cases} 0 & \text{if } j=0 \\ \underbrace{\max \{ v_j + OPT(p(j))}_{\text{Case 1}},\ \underbrace{OPT(j-1) \}}_{\text{Case 2}} & \text{otherwise} \end{cases}$$

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty   ← global array
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(w_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Schduling

Sequence Alignment: time:O(mn) space:O(mn) to space:O(m+n). **Shortest Paths:** Θ(mn) time.Bellman-ford algorithm.Detecting Negative Cycles: If OPT(n,v) = OPT(n-1,v) for all v, then there is no negative cycle with a path to t. We can use Bellman-ford to detect negative cost cycle in O(mn) time.

- **Def.** OPT(i, j) = min cost of aligning strings $x_1x_2…x_i$ and $y_1y_2…y_j$
- Case 1: OPT matches $x_i - y_j$
  - pay mismatch for $x_i - y_j$ + min cost of aligning two strings $x_1x_2…x_{i-1}$ and $y_1y_2…y_{j-1}$
- Case 2a: OPT leaves x_i unmatched
  - pay gap for x_i and min cost of aligning $x_1x_2…x_{i-1}$ and $y_1y_2…y_j$
- Case 2b: OPT leaves y_j unmatched
  - pay gap for y_j and min cost of aligning $x_1x_2…x_i$ and $y_1y_2…y_{j-1}$

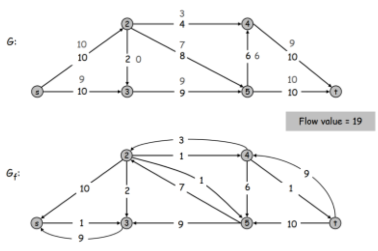
$$OPT(i,j)=\begin{cases} j\delta & \text{if } i=0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j=0 \end{cases}$$

```
Sequence-Alignment(m, n, x_1x_2...x_m, y_1y_2...y_n, δ, α) {
   for i = 0 to m
      M[0, i] = iδ
   for j = 0 to n
      M[j, 0] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[x_i, y_j] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

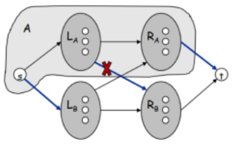
Analysis: Θ(mn) time and space

```
Shortest-Path(G, s, t) {
   foreach node v ∈ V
      M[0, v] ← ∞
   M[0, t] ← 0

   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] ← M[i-1, v]
      foreach edge (v, w) ∈ E
         M[i, v] ← min { M[i, v], M[i-1, w] + c_vw }

   return M[n-1, s]
}
```

Network Flow: **Flow network:** Abstraction for material flowing through the edges. G = (V, E) = directed graph. Two distinguished nodes: s = source, t = sink. c(e) = nonnegative capacity of edge e. **s-t flow:** a function that satisfies: For each e∈E: 0<=f(e)<=c(e)(capacity), For each v∈V – {s, t}: f(e)(e in to v)= f(e)(e out of v). The value of a flow f is: f(e)(e out of s). **Max flow problem:** Find s-t flow of maximum value. **Residual graph:** G f= (V, E f), Residual edges with positive residual capacity, E_f={e:f(e)<c(e)}∪{e^R:f(e)>0}. **s-t cut:** a partition (A, B) of V with s∈A and t∈B, The capacity of a cut (A, B) is: c(e)(e out of A). **Min s-t cut problem:** Find an s-t cut of minimum capacity. **Marriage Theorem:** G has a perfect matching iff |N(S)|>=|S| for all subsets S ⊆L. -> This was the previous observation.
Flow value lemma: Let f be any flow, and let (A, B) be any s-t cut. Then, f(e)(e out of A)-f(e)(e in to A)=Value(f).
Weak duality: Let f be any flow. Then, for any s-t cut (A, B) we have v(f) <= cap(A, B).

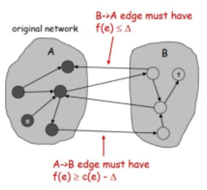


Pf. ← Suppose G does not have a perfect matching
- Formulate as a max flow problem and let (A, B) be min cut in G′
- Define $L_A = L∩A$, $L_B = L∩B$, $R_A = R∩A$, $R_B = R∩B$
- Cap(A, B) = v(f*) = |M| < |L| ("<": because no perfect matching)
- Since min cut can't use ∞ edges, no edge between L_A and R_B
  - Cap(A, B) = $|L_B| + |R_A|$
  - $N(L_A) ⊆ R_A$
- $|N(L_A)| <= |R_A|$
  $= cap(A, B) - |L_B|$
  $< |L| - |L_B|$
  $= |L_A|$
- This contradicts the condition



Pf. (almost identical to proof of max-flow min-cut theorem)
- We show that at the end of a Δ-phase, there exists a cut (A, B) such that cap(A, B) <= v(f) + mΔ
- Choose A to be the set of nodes reachable from s in $G_f(Δ)$
- By definition of A, s ∈ A
- By definition of f, t ∉ A

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &\geq \sum_{e \text{ out of } A} (c(e)-\Delta) - \sum_{e \text{ in to } A} \Delta \\ &= \sum_{e \text{ out of } A} c(e) - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ in to } A} \Delta \\ &\geq cap(A,B) - m\Delta \quad \blacksquare \end{aligned}$$


original network
B->A edge must have f(e) ≤ Δ
A
B
s
t
A->B edge must have f(e) ≥ c(e) - Δ

Max-flow min-cut theorem: The value of the max flow is equal to the value of the min cut.
Ford-Fulkerson Algorithm: Choosing Good Augmenting Paths. Choosing path with high bottleneck capacity. Δ-residual graph Gf(Δ). Δ取 2 的幂小于 C. Capacity Scaling Running Time: •Lemma 1. The outer while loop repeats 1+log2 C times. Pf. Initially C/2 < Δ <= C. Δ decreases by a factor of 2 each iteration. • Lemma 2. Let f be the flow at the end of a Δ-scaling phase. Then the value of the maximum flow is at most v(f) + mΔ. •Lemma 3. There are at most 2m augmentation per scaling phase.Let f be the flow at then end of the previous scaling phase. L2-> V(f*) <= v(f) + m(2Δ). Each augmentation in a Δ-phase increases v(f) by at least Δ. The scaling max-flow algorithm finds a max flow in O(mlogC) augmentations. It can be implemented to run in O(m^2 logC) time.

Pf.

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

by flow conservation, all terms except v = s are 0 $\longrightarrow$

$$= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to } v} f(e) \right)$$

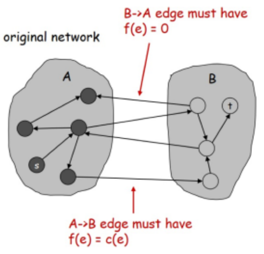
$$= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e).$$

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &\le \sum_{e \text{ out of } A} f(e) \\ &\le \sum_{e \text{ out of } A} c(e) \\ &= \text{cap}(A, B) \quad ▪ \end{aligned}$$

- **Proof strategy.** We prove both simultaneously by showing the equivalence of the following three conditions for any flow f:
  - (i) There exists a cut (A, B) such that v(f) = cap(A, B)
  - (ii) Flow f is a max flow
  - (iii) There is no augmenting path relative to f
- (i) → (ii) This was the corollary to weak duality lemma
- (ii) → (iii) We show contrapositive
  - If there exists an augmenting path, then we can improve f by sending flow along path
- (iii) → (i)
  - Let f be a flow with no augmenting paths
  - Let A be set of vertices reachable from s in residual graph
  - By definition of A, s ∈ A
  - By definition of f, t ∉ A

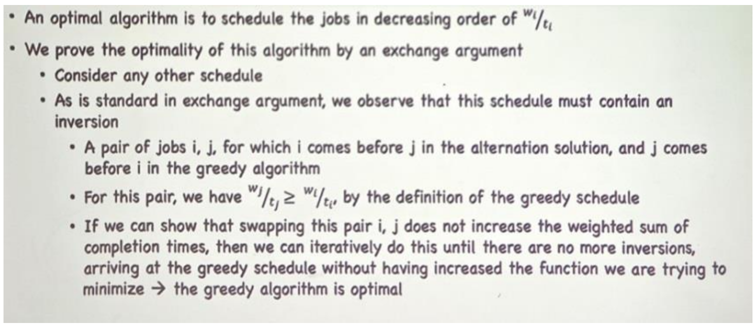
$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &= \sum_{e \text{ out of } A} c(e) \quad ▪ \end{aligned}$$

original network

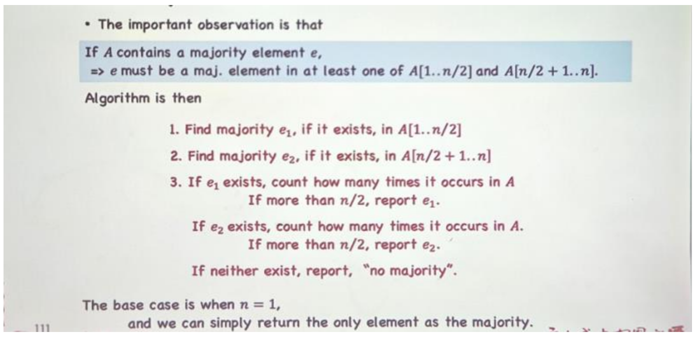


```
Push-Based-Shortest-Path(G, s, t) {
   foreach node v ∈ V
      M[v] ← ∞
   M[t] ← 0

   for i = 1 to n-1 {
      foreach node w ∈ V
         if (M[w] has been updated in previous iteration)
            foreach node v such that (v, w) ∈ E
               if (M[v] > M[w] + c_vw)
                  M[v] ← M[w] + c_vw
      If no M[w] value changed in iteration i, stop.
   }

   return M[s]
}
```

- Add new node t and connect all nodes to t with 0-cost edge
- Check if OPT(n, v) = OPT(n-1, v) for all nodes v
  - If yes, then no negative cycles
  - If no, then extract cycle from shortest path from v to t



- An optimal algorithm is to schedule the jobs in decreasing order of w_i/t_i
- We prove the optimality of this algorithm by an exchange argument
  - Consider any other schedule
  - As is standard in exchange argument, we observe that this schedule must contain an inversion
    - A pair of jobs i, j, for which i comes before j in the alternation solution, and j comes before i in the greedy algorithm
    - For this pair, we have $w_j/t_j \ge w_i/t_i$, by the definition of the greedy schedule
    - If we can show that swapping this pair i, j does not increase the weighted sum of completion times, then we can iteratively do this until there are no more inversions, arriving at the greedy schedule without having increased the function we are trying to minimize → the greedy algorithm is optimal



- The important observation is that

If A contains a majority element e,
=> e must be a maj. element in at least one of $A[1..n/2]$ and $A[n/2+1..n]$.

Algorithm is then

1. Find majority e_1, if it exists, in $A[1..n/2]$
2. Find majority e_2, if it exists, in $A[n/2+1..n]$
3. If e_1 exists, count how many times it occurs in A
   If more than $n/2$, report e_1.

   If e_2 exists, count how many times it occurs in A.
   If more than $n/2$, report e_2.

   If neither exist, report, "no majority".

The base case is when $n = 1$,
and we can simply return the only element as the majority.

111

- Greedy template: Consider jobs in some order
- [Shortest processing time first] Consider jobs in ascending order of processing time t_j

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Earliest deadline first] Consider jobs in ascending order of deadline d_j
- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$

	1	2
t_j	1	10
d_j	2	10

counterexample