Huffman code:

HUFFMAN(C)
$n = |C|$
$Q = C$
for $i = 1$ **to** $n - 1$
  allocate a new node z
  $z.left = x =$ EXTRACT-MIN(Q)
  $z.right = y =$ EXTRACT-MIN(Q)
  $z.freq = x.freq + y.freq$
  INSERT(Q, z)
return EXTRACT-MIN(Q) // return the roo

O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2*(n – 1) times. extractMin() takes O(logn) time as it calles minHeapify(). So, overall complexity is O(nlogn).
If the input array is sorted, there exists a linear time algorithm.

Graph 表达：

G(V,E): v: 端点, e: 路径

Adj-list-represent: consists of an array of each vertex, like Adj[u] = all the vertices v such that there is an edge(u,v) in E. for directed graph, the sum of length of all the adj list is |E|, for undirected graph, is |2E|. the amount of memory is Θ(E+V).

Another: adj-matrix-representation:
a_{ij} = 1 if (i,j) in E, 0 otherwise

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

The Bellman Ford algorithm:
Dynamic programing base-
Subgame-property:

Lemma 24.1 (Subpaths of shortest paths are shortest paths)
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

Proof If we decompose path p into $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$, then we have that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p'_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ is a path from v_0 to v_k whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k. ∎

initialize:

INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex $v \in G.V$
  $v.d = \infty$
  $v.\pi = \text{NIL}$
$s.d = 0$

relax:

RELAX(u, v, w)
if $v.d > u.d + w(u, v)$
  $v.d = u.d + w(u, v)$
  $v.\pi = u$

包含 negative-edge 没事, but can't include negative-circle.

BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 **for** $i = 1$ **to** $|G.V| - 1$
3   **for** each edge $(u, v) \in G.E$
4   RELAX(u, v, w)
5 **for** each edge $(u, v) \in G.E$
6   **if** $v.d > u.d + w(u, v)$
7   **return** FALSE
8 **return** TRUE

2-4 update the s-p, 5-8 check negative circle.

```
# The main function that finds shortest distances from src to
# all other vertices using Bellman-Ford algorithm.  The function
# also detects negative weight cycle
def BellmanFord(self, src):

    # Step 1: Initialize distances from src to all other vertices
    # as INFINITE
    dist = [float("Inf")] * self.V
    dist[src] = 0


    # Step 2: Relax all edges |V| - 1 times. A simple shortest
    # path from src to any other vertex can have at-most |V| - 1
    # edges
    for i in range(self.V - 1):
        # Update dist value and parent index of the adjacent vertices of
        # the picked vertex. Consider only those vertices which are still in
        # queue
        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

    # Step 3: check for negative-weight cycles.  The above step
    # guarantees shortest distances if graph doesn't contain
    # negative weight cycle.  If we get a shorter path, then there
    # is a cycle.

    for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    print "Graph contains negative weight cycle"
                    return

    # print all distance
    self.printArr(dist)
```

The Bellman-Ford algorithm runs in time O(VE), since the initialization in line 1 takes (V) time, each of the |V|-1 passes over the edges in lines 2–4, takes (E) time, and the for loop of lines 5–7 takes O(E)time.

Check how many loop:

Before each iteration of the for loop on line 2, we make a backup copy of the current d values for all the vertices. Then, after each iteration, we check to see if any of the d values changed. If none did, then we immediately terminate the for loop. This clearly works because if one iteration didn't change the values of d, nothing will of changed on later iterations, and so they would all proceed to not change any of the d values.

Dijkstra's shortest path algorithm:
running time: V^2 but can be O(ElgV) if we implement the binary min-heap.

DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
  $u =$ EXTRACT-MIN(Q)
  $S = S \cup \{u\}$
  for each vertex $v \in G.Adj[u]$
    RELAX(u, v, w)

Algorithm 2 RELIABILITY(G, r, x, y)
INITIALIZE-SINGLE-SOURCE(G, x)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$ **do**
  $u =$ EXTRACT-MIN(Q)
  $S = S \cup \{u\}$
  for each vertex $v \in G.Adj[u]$ **do**
    if $v.d < u.d \cdot r(u, v)$ **then**
      $v.d = u.d \cdot r(u, v)$
      $v.\pi = u$
    end if
  end for
end while
while $y \neq x$ **do**
  Print y
  $y = y.\pi$
end while
Print x

All-pairs-shortest-path:

Time $= \Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

For Dijkstra, with linear-array, time is $O(V^3)$, if with binary-min-heap, the time is O(VE*logV).

If contains negative edge, we only can use bellman-ford:
The time is $O(V^2E) = O(V^4)$ for a dense graph, since $E = V^2$.

Sub-game:

$$l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)}, \min_{1 \le k \le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) = \min_{1 \le k \le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} .$$

bottom-up: Given a W and $L^{(m-1)}$, return the $L^{(m)}$

EXTEND-SHORTEST-PATHS(L, W)
$n = L.rows$
let $L' = (l'_{ij})$ be a new $n \times n$ matrix
for $i = 1$ **to** n
  for $j = 1$ **to** n
    $l'_{ij} = \infty$
    for $k = 1$ **to** n
      $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$
return L'

这个算法的实际相当于 matrix multiply.
So using extend-method, to calculate the cost:

SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
$n = W.rows$
$L^{(1)} = W$
for $m = 2$ **to** $n - 1$
  let $L^{(m)}$ be a new $n \times n$ matrix
  $L^{(m)} =$ EXTEND-SHORTEST-PATHS($L^{(m-1)}, W$)
return $L^{(n-1)}$

we only need to calculate $L^{(n-1)}$, and instead of adding by 1, we can multiple by 2.

FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
$n = W.rows$
$L^{(1)} = W$
$m = 1$
while $m < n - 1$
  let $L^{(2m)}$ be a new $n \times n$ matrix
  $L^{(2m)} =$ EXTEND-SHORTEST-PATHS($L^{(m)}, L^{(m)}$)
  $m = 2m$
return $L^{(m)}$

Because each of the lg(n-1) matrix products takes n^3 time, FASTERALL-PAIRS-SHORTEST-PATHS runs in (n^3lg n) time.

To verify associativity, we need to check that $(W^iW^j)W^p = W^i(W^jW^p)$ for all i, j, p, where we use the matrix multiplication defined by the EXTEND-SHORTEST-PATHS procedure. Consider entry (a, b) of the left hand side. This is:

$$\begin{aligned} \min_{1 \le k \le n} [W^iW^j]_{a,k} + W^p_{k,b} &= \min_{1 \le k \le n} \min_{1 \le q \le n} W^i_{a,q} + W^j_{q,k} + W^p_{k,b} \\ &= \min_{1 \le q \le n} W^i_{a,q} + \min_{1 \le k \le n} W^j_{q,k} + W^p_{k,b} \\ &= \min_{1 \le q \le n} W^i_{a,q} + [W^jW^p]_{q,b} \end{aligned}$$

which is precisely entry (a, b) of the right hand side.

computer the vertices on the S-Paths:

Algorithm 1 EXTEND-SHORTEST-PATH-MOD(II, L,W)
n= L.rows
Let $L' = (l'_{ij})$ be a new $n \times n$ matrix.
$\Pi' = (\pi'_{ij})$ is a new $n \times n$ matrix
for i=1 to n **do**
  for j = 1 to n **do**
    $l'_{ij} = \infty$
    $\pi_{ij} = NIL$
    for k=1 to n **do**
      if $l_{ik} + l_{jk} < l_{ij}$ **then**
        $l_{ij} = l_{ik} + l_{jk}$
        $\pi'_{ij} = \pi_{kj}$
      end if
    end for
  end for
end for
return Π', L'

Algorithm 2 SLOW-ALL-PAIRS-SHORTEST-PATHS-MOD(W)
n= W.rows
$L^{(1)} = W$
$\Pi^{(1)} = (\pi^{(1)}_{ij})$ where $\pi^{(1)}_{ij} = i$ if there is an edge from i to j, and NIL otherwise.
for m=2 to n-1 **do**
  $\Pi^{(m)}, L^{(m)} =$ EXTEND-SHORTEST-PATH-MOD($\Pi^{(m-1)}, L^{(m-1)}$,W)
end for
return $\Pi^{(n-1)}, L^{(n-1)}$

when the memory require is $O(n^2)$:

We can overwrite matrices as we go. Let $A \star B$ denote multiplication defined by the EXTEND-SHORTEST-PATHS procedure. Then we modify FASTER-ALL-EXTEND-SHORTEST-PATHS(W). We initially create an n by n matrix L. Delete line 5 of the algorithm, and change line 6 to $L = W \star W$, followed by $W = L$.

Floyd-Warshall-all pairs-sp:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

```
FLOYD-WARSHALL(W)
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

calculate the predecessor matrix from the completed matrix L is O(n[3]) time.

For each source vertex v_i we need to compute the shortest-paths tree for v_i. To do this, we need to compute the predecessor for each $j \neq i$. For fixed i and j, this is the value of k such that $L_{i,k} + w(k,j) = L_{i,j}$. Since there are n vertices whose trees need computing, n vertices for each such tree whose predecessors need computing, and it takes $O(n)$ to compute this for each one (checking each possible k), the total time is $O(n^3)$.

constructing a shortest path:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\,, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\,. \end{cases}$$

Algorithm 3 MOD-FLOYD-WARSHALL(W)

```
n= W.rows
D^0 = W
π^0 is a matrix with nil in every entry
for i=1 to n do
    for j = 1 to n do
        if i ≠ j and D^0_{i,j} < ∞ then
            π^0_{i,j} = i
        end if
    end for
end for
for k=1 to n do
    let D^k be a new n × n matrix.
    let π^k be a new n × n matrix
    for i=1 to n do
        for j = 1 to n do
            if d^{k-1}_{ij} ≤ d^{k-1}_{i,k} + d^{k-1}_{k,j} then
                d^k_{i,j} = d^{k-1}_{i,j}
                π^k_{i,j} = π^{k-1}_{i,j}
            else
                d^k_{i,j} = d^{k-1}_{i,k} + d^{k-1}_{k,j}
                π^k_{i,j} = π^k_{k,j}
            end if
        end for
    end for
end for
```

determine whether there is a path from a to b:

We set $w_{ij} = 1$ if (i,j) is an edge, and $w_{ij} = 0$ otherwise. Then we replace line 7 of EXTEND-SHORTEST-PATHS(L,W) by $l'_{ij} = l'_{ij} \vee (l_{ik} \wedge w_{kj})$. Then run the SLOW-ALL-PAIRS-SHORTEST-PATHS algorithm.

with space O(n[2]) required of F-W:

```
FLOYD-WARSHALL'(W)
n = W.rows
D = W
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            d_ij = min(d_ij, d_ik + d_kj)
return D
```

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(by equation (5.2))} \quad (5.4)\\ &= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(by linearity of expectation)}\\ &= \sum_{i=1}^{n} 1/i && \text{(by equation (5.3))}\\ &= \ln n + O(1) && \text{(by equation (A.7))}\,. \quad (5.5) \end{aligned}$$

Even though we interview n people, we actually hire only approximately $\ln n$ of them, on average. We summarize this result in the following lemma.

5.2-1

In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly one time? What is the probability that you hire exactly n times?

You will hire exactly one time if the best candidate is presented first. There are $(n-1)!$ orderings with the best candidate first, so, it is with probability $\frac{(n-1)!}{n!} = \frac{1}{n}$ that you only hire once. You will hire exactly n times if the candidates are presented in increasing order. This fixes the ordering to a single one, and so this will occur with probability $\frac{1}{n!}$.

Let $X_{i,j}$ for $i < j$ be the indicator of $A[i] > A[j]$. Then, we have that the expected number of inversions is

$$E\left[\sum_{i<j} X_{i,j}\right] = \sum_{i<j} E[X_{i,j}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr(A[i] > A[j]) = \frac{1}{2}\sum_{i=1}^{n-1} n - i$$
$$= \frac{n(n-1)}{2} - \frac{n(n-1)}{4} = \frac{n(n-1)}{4}.$$

randomize-quick-sort: same time complexity

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

The new quicksort calls RANDOMIZED-PARTITION

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

randomize 的实质: balance the recursive tree. So the depth of recursion tree is lg(n), so the total time for this is O(nlgn).

$$\begin{aligned} \Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}\\ &= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\}\\ &\quad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\}\\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1}\\ &= \frac{2}{j-i+1}\,. \end{aligned}$$

$$\mathrm{E}[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1}\,.$$

We can evaluate this sum using a cha on the harmonic series in equation (A

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1}\\ &= \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1}\\ &< \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k}\\ &= \sum_{i=1}^{n-1} O(\lg n)\\ &= O(n \lg n)\,. \end{aligned}$$

randomize-q-sort:

worst-case:

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n)\,, \quad (7.1)$$

where the parameter q ranges from 0 to $n-1$ because the procedure PARTITION produces two subproblems with total size $n-1$. We guess that $T(n) \leq cn^2$ for some constant c. Substituting this guess into recurrence (7.1), we obtain

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} (cq^2 + c(n-q-1)^2) + \Theta(n)\\ &= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n-q-1)^2) + \Theta(n)\,. \end{aligned}$$

observation gives us the bound $\max_{0 \leq q \leq n-1}(q^2 + (n-q-1)^2) \leq (n-1)^2 = n^2 - 2n + 1$. Continuing with our bounding of $T(n)$, we obtain

$$\begin{aligned} T(n) &\leq cn^2 - c(2n-1) + \Theta(n)\\ &\leq cn^2\,, \end{aligned}$$

best-case, view q =n/2: Θ(nlgn):

$$T(n) = \min_{1 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n).$$

$$\begin{aligned} T(n) &\geq \min_{1 \leq q \leq n-1} (cq\lg q + 2cq + c(n-q-1)\lg(n-q-1) + 2c(n-q-1)) + \Theta(n)\\ &= \frac{cn}{2}\lg(n/2) + cn + c(n/2-1)\lg(n/2-1) + cn - 2c + \Theta(n)\\ &\geq (cn/2)\lg n - cn/2 + c(n/2-1)(\lg n - 2) + 2cn - 2c\Theta(n)\\ &= (cn/2)\lg n - cn/2 + (cn/2)\lg n - cn - \lg n + 2 + 2cn - 2c\Theta(n)\\ &= cn\lg n + cn/2 - \lg n + 2 - 2c + \Theta(n) \end{aligned}$$

排列 A(n,m)=n×（n-1）....(n-m+1) =n! /(n-m)!

组合:C(n,m)=P(n,m)/P(m,m) =n!/m! *(n-m)！

```
RANDOMIZE-IN-PLACE(A)
n = A.length
for i = 1 to n
    swap A[i] with A[RANDOM(i, n)]
```

proof: e1 = the previous loop, e12= ith iteration

$$\begin{aligned} \Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\}\Pr\{E_1\}\\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!}\\ &= \frac{(n-i)!}{n!}\,. \end{aligned}$$

when using with-all = you got 27 possible end states, but only 6 possible ordering, so probability is not equal.

Find i[th] element:

```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k = q - p + 1
if i == k           // the pivot value is the answer
    return A[q]
elseif i < k
    return RANDOMIZED-SELECT(A, p, q - 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

Algorithm 1 ITERATIVE-RANDOMIZED-SELECT

```
while p < r do
    q = RANDOMIZED − PARTITION(A, p, r)
    k = q − p + 1
    if i=k then
        return A[q]
    end if
    if i < k then
        r = q − 1
    else
        p = q
        i = i − k
    end if
end while
return A[p]
```

9.2-4

Suppose we use RANDOMIZED-SELECT to select the minimum element of the array $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

Exercise 9.2-4

When the partition selected is always the maximum element of the array we get worst-case performance. In the example, the sequence would be 9, 8, 7, 6, 5, 4, 3, 2, 1, 0.

hat-check problem: someone get his hat back:

Let X be the number of customers who get back their own hat and X_i be the indicator random variable that customer i gets his hat back. The probability that an individual gets his hat back is $\frac{1}{n}$. Then we have

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{n} = 1.$$

5.2-5

Let $A[1 \ldots n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A. (See Problem 2-4 for more on inversions.) Suppose that the elements of A form a uniform random permutation of $\langle 1, 2, \ldots, n \rangle$. Use indicator random variables to compute the expected number of inversions.

Let $X_{i,j}$ for $i < j$ be the indicator of $A[i] > A[j]$. Then, we have that the expected number of inversions is

$$E\left[\sum_{i<j} X_{i,j}\right] = \sum_{i<j} E[X_{i,j}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr(A[i] > A[j]) = \frac{1}{2}\sum_{i=1}^{n-1} n - i$$
$$= \frac{n(n-1)}{2} - \frac{n(n-1)}{4} = \frac{n(n-1)}{4}.$$