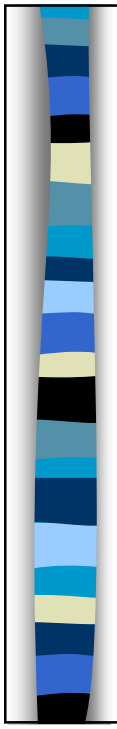
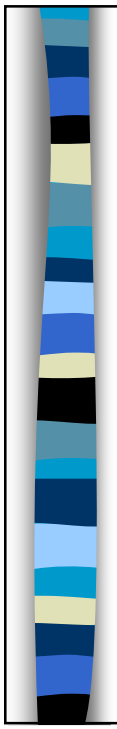


## Divide-and-Conquer

The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions



## Randomized Algorithm

- An algorithm whose behaviour depends on:
  - an input
  - a sequence of random bits
- Can be faster and simpler than a deterministic (no random bits) algorithm
- Two types:
  - Las Vegas
  - Monte Carlo

## Las Vegas and Monte Carlo

- Named after famous casinos
- Las Vegas:
  - always returns correct answer
  - running time fast with a high probability
  - "everyone wins in Vegas"
- Monte Carlo:
  - always runs fast
  - returns correct answer with a high probability

## Primality Test

- Fermat's Little Theorem:
  - If n is prime, then for all $a$ in [1,$n$-1],
    - $a^n - a = 0 \pmod{n}$
  - E.g. $4^3 - 4 = 60$ is divisible by 3
- First attempt at algorithm:
  1. Choose $a$ at random from [2,$n$-1]
  2. If $a^n - a = 0 \pmod{n}$, then return COMPOSITE
  3. Else return PRIME

# Primality Test

- However, there are exceptions to Fermat's Little Theorem, called the **Carmichael numbers**
- The previous algorithm returns PRIME for the Carmichael numbers
- A modification to the algorithm yields the **Miller-Rabin test**, which bounds the probability of running into a Carmichael number at 0.5

# Miller-Rabin – Proof of Bound

If $n$ is an odd composite number, then the number of witnesses to the compositeness of $n$ is at least $(n - 1)/2$.

*Proof* The proof shows that the number of nonwitnesses is at most $(n - 1)/2$, which implies the theorem.

We start by claiming that any nonwitness must be a member of $\mathbb{Z}_n^*$. Why? Consider any nonwitness $a$. It must satisfy $a^{n-1} \equiv 1 \pmod{n}$ or, equivalently, $a \cdot a^{n-2} \equiv 1 \pmod{n}$. Thus, there is a solution to the equation $ax \equiv 1 \pmod{n}$, namely $a^{n-2}$. By Corollary 31.21, $\gcd(a, n) = 1$, which in turn implies that $\gcd(a, n) = 1$. Therefore, $a$ is a member of $\mathbb{Z}_n^*$; all nonwitnesses belong to $\mathbb{Z}_n^*$.

To complete the proof, we show that not only are all nonwitnesses contained in $\mathbb{Z}_n^*$, they are all contained in a proper subgroup $B$ of $\mathbb{Z}_n^*$ (recall that we say $B$ is a *proper* subgroup of $\mathbb{Z}_n^*$ when $B$ is a subgroup of $\mathbb{Z}_n^*$ but $B$ is not equal to $\mathbb{Z}_n^*$). By Corollary 31.16, we then have $|B| \le |\mathbb{Z}_n^*|$. Since $|\mathbb{Z}_n^*| \le n - 1$, we obtain $|B| \le (n - 1)/2$. Therefore, the number of nonwitnesses is at most $(n - 1)/2$, so that the number of witnesses must be at least $(n - 1)/2$.

We now show how to find a proper subgroup $B$ of $\mathbb{Z}_n^*$ containing all of the nonwitnesses. We break the proof into two cases.

*Case 1:* There exists an $x \in \mathbb{Z}_n^*$ such that

$$x^{n-1} \not\equiv 1 \pmod{n}.$$

In other words, $n$ is not a Carmichael number. Because, as we noted earlier, Carmichael numbers are extremely rare, case 1 is the main case that arises "in practice" (e.g., when $n$ has been chosen randomly and is being tested for primality).

Let $B = \{x \in \mathbb{Z}_n^* : x^{n-1} \equiv 1 \pmod{n}\}$. Clearly, $B$ is nonempty, since $1 \sqsubset B$. Since $B$ is closed under multiplication modulo $n$, we have that $B$ is a subgroup of $\mathbb{Z}_n^*$ by Theorem 31.14. Note that every nonwitness belongs to $B$, since a nonwitness $a$ satisfies $a^{n-1} \equiv 1 \pmod{n}$. Since $x \in \mathbb{Z}_n^* - B$, we have that $B$ is a proper subgroup of $\mathbb{Z}_n^*$.

*Case 2:* For all $x \in \mathbb{Z}_n^*$,

$$(31.39)\quad x^{n-1} \equiv 1 \pmod{n}.$$

In other words, $n$ is a Carmichael number. This case is extremely rare in practice. However, the Miller-Rabin test (unlike a pseudo-primality test) can efficiently determine the compositeness of Carmichael numbers, as we now show.

In this case, $n$ cannot be a prime power. To see why, let us suppose to the contrary that $n = p^e$, where $p$ is a prime and $e > 1$. We derive a contradiction as follows. Since $n$ is assumed to be odd, $p$ must also be odd. Theorem 31.32 implies that $\mathbb{Z}_n^*$ is a cyclic group; it contains a generator $g$ such that $\mathrm{ord}_n(g) = |\mathbb{Z}_n^*| = \phi(n) = p^{e-1}(p - 1)(n = (p - 1)p^{e-1})$. By equation (31.39), we have $g^{n-1} \equiv 1 \pmod{n}$. Then the discrete logarithm theorem (Theorem 31.33, taking $y = 0$) implies that $n - 1 \equiv 0 \pmod{\phi(n)}$, or

$$(p - 1)p^{e-1} \mid p^e - 1.$$

This is a contradiction for $e > 1$, since $(p - 1)p^{e-1}$-1 is divisible by the prime $p$ but $p^e - 1$ is not. Thus, $n$ is not a prime power.

Since the odd composite number $n$ is not a prime power, we decompose it into a product $n_1 n_2$, where $n_1$ and $n_2$ are odd numbers greater than 1 that are relatively prime to each other. (There may be several ways to do this, and it doesn't matter which one we choose. For example, if $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, then we can choose $n_1 = p_1^{e_1}$ and $n_2 = p_2^{e_2} p_3^{e_3} \cdots p_r^{e_r}$.)

Recall that we define $t$ and $u$ so that $n - 1 = 2^t u$, where $t \ge 1$ and $u$ is odd, and that for an input $a$, the procedure WITNESS computes the sequence

$$X = \langle a^u, a^{2u}, a^{2^2 u}, \ldots, a^{2^t u} \rangle$$

(all computations are performed modulo $n$).

Let us call a pair $(v, j)$ of integers *acceptable* if $v \in \mathbb{Z}_n$, $j \sqsubset \{0, 1, \ldots, t\}$, and

$$v^{2^j} \equiv -1 \pmod{n}.$$

Acceptable pairs certainly exist since $u$ is odd; we can choose $v = n - 1$ and $j = 0$, so that $(n - 1, 0)$ is an acceptable pair. Now pick the largest possible $j$ such that there exists an acceptable pair $(v, j)$, and fix $v$ so that $(v, j)$ is an acceptable pair. Let

$$B = \{x \in \mathbb{Z}_n^* : x^{2^j v} \equiv \pm 1 \pmod{n}\}.$$

Since $B$ is closed under multiplication modulo $n$, it is a subgroup of $\mathbb{Z}_n^*$. By Corollary 31.16, therefore, $|B|$ divides $|\mathbb{Z}_n^*|$. Every nonwitness must be a member of $B$, since the sequence $X$ produced by a nonwitness must either be all 1's or else contain a -1 no later than the $j$th position, by the maximality of $j$. ($\exists s (a, j)$ is acceptable, where $a$ is a nonwitness, we must have $j \le j$ by how we chose $j$.)

We now use the existence of $v$ to demonstrate that there exists a $w \in \mathbb{Z}_n - B$. Since $v^{2^j} = -1 \pmod{n}$, we have $v^{2^j} \equiv -1 \pmod{n_1}$ by Corollary 31.28 to the Chinese remainder theorem. By Corollary 31.28, there is a $w$ simultaneously satisfying the equations

$$w \equiv v \pmod{n_1},$$
$$w \equiv 1 \pmod{n_2}.$$

Therefore,

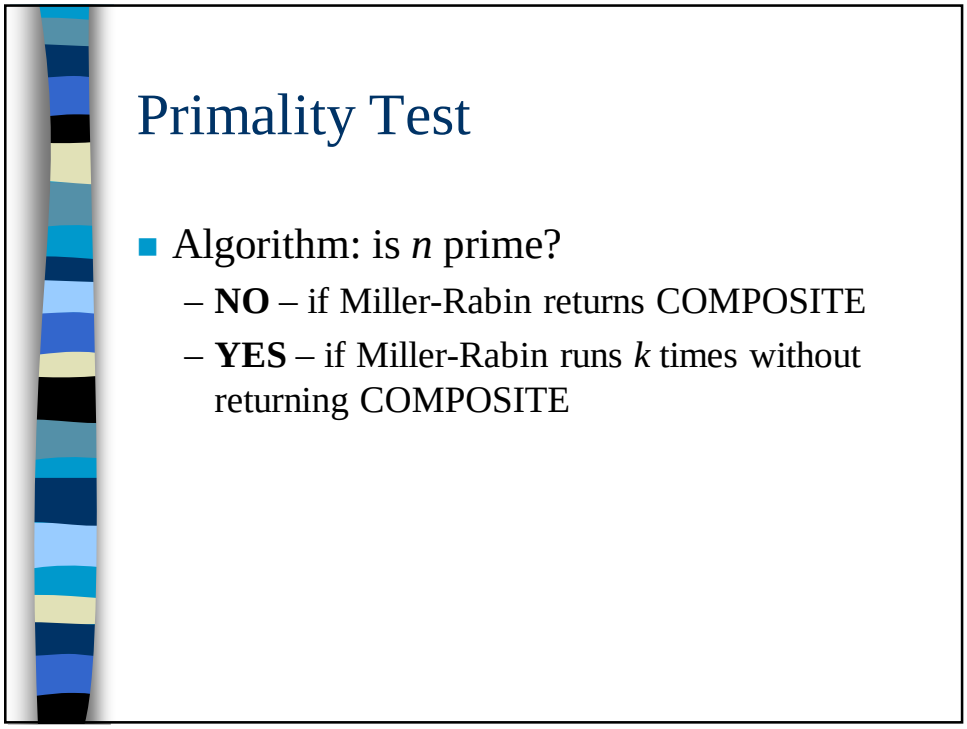
$$w^{2^j} \equiv -1 \pmod{n_1},$$
$$w^{2^j} \equiv 1 \pmod{n_2}.$$

By Corollary 31.29, $w^{2^j v} \not\equiv 1 \pmod{n_1}$ implies $w^{2^j v} \not\equiv 1 \pmod{n}$, and $w^{2^j v} \not\equiv -1 \pmod{n_2}$ implies $w^{2^j v} \not\equiv -1 \pmod{n}$. Hence, $w^{2^j v} \not\equiv \pm 1 \pmod{n}$, and so $w \notin B$.

It remains to show that $w \in \mathbb{Z}_n^*$, which we do by first working separately modulo $n_1$ and modulo $n_2$. Working modulo $n_1$, we observe that since $v \in \mathbb{Z}_n^*$, we have that $\gcd(v, n) = 1$, and so $\gcd(v, n_1) = 1$; if $v$ doesn't have any common divisors with $n$, it certainly doesn't have any common divisors with $n_1$. Since $w \equiv v \pmod{n_1}$, we see that $\gcd(w, n_1) = 1$. Working modulo $n_2$, we observe that $w \equiv 1 \pmod{n_2}$ implies $\gcd(w, n_2) = 1$. To combine these results, we use Theorem 31.6, which implies that $\gcd(w, n_1 n_2) = \gcd(w, n) = 1$. That is, $w \in \mathbb{Z}_n^*$.

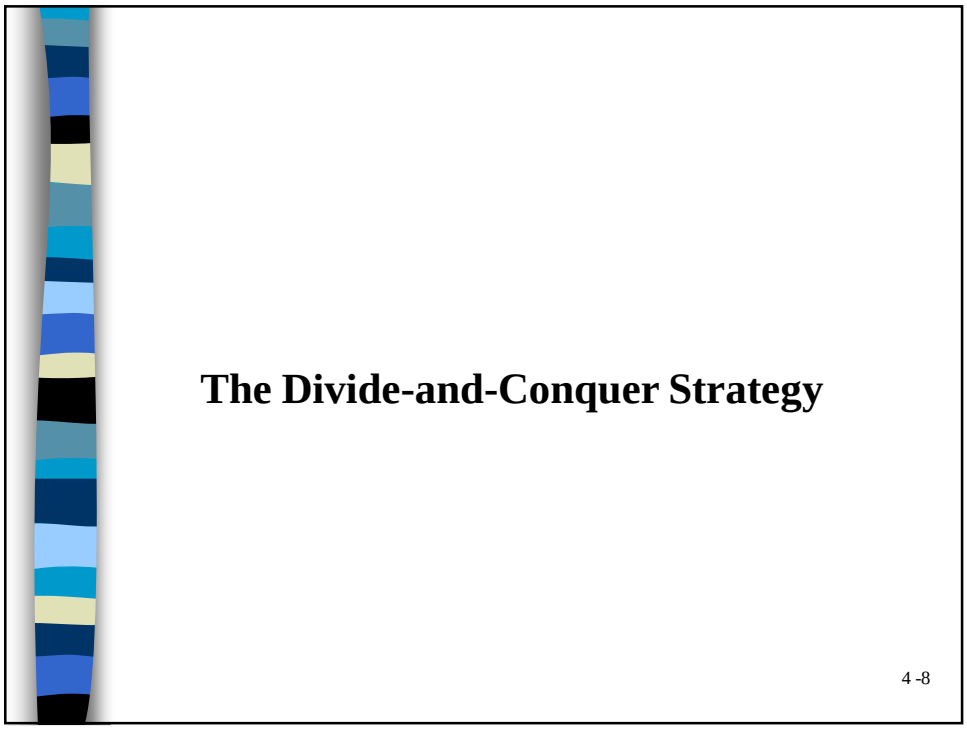
Therefore $w \in \mathbb{Z}_n^* - B$, and we finish case 2 with the conclusion that $B$ is a proper subgroup of $\mathbb{Z}_n^*$.

In either case, we see that the number of witnesses to the compositeness of $n$ is at least $(n - 1)/2$.



# Primality Test

- Algorithm: is *n* prime?
  - **NO** – if Miller-Rabin returns COMPOSITE
  - **YES** – if Miller-Rabin runs *k* times without returning COMPOSITE

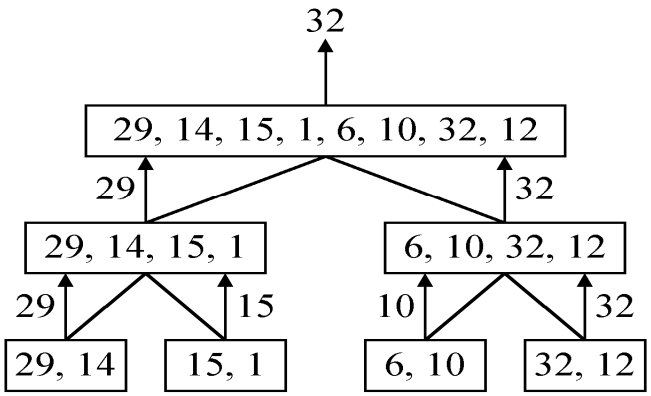


**The Divide-and-Conquer Strategy**

4 -8

# A simple example

- finding the maximum of a set S of n numbers




4 -9

# Time complexity

- Time complexity:

T(n): # of comparisons

$$T(n)=\begin{cases} 2T(n/2)+1 & , n>2 \\ 1 & , n\leq 2 \end{cases}$$

- Calculation of T(n):

Assume $n = 2^k$,

$T(n) = 2T(n/2)+1$

$= 2(2T(n/4)+1)+1$

$= 4T(n/4)+2+1$

:

$=2^{k-1}T(2)+2^{k-2}+\ldots+4+2+1$

$=2^{k-1}+2^{k-2}+\ldots+4+2+1$

$=2^k-1 = n-1$

4 -10

# A general divide-and-conquer algorithm

Step 1: If the problem size is small, solve this problem directly; otherwise, split the original problem into 2 sub-problems with equal sizes.

Step 2: Recursively solve these 2 sub-problems by applying this algorithm.

Step 3: Merge the solutions of the 2 sub-problems into a solution of the original problem.

4 -11

# Time complexity of the general algorithm

- Time complexity:

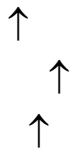
$$T(n)=\begin{cases} 2T(n/2)+S(n)+M(n) & , n \geq c \\ b & , n < c \end{cases}$$

where S(n) : time for splitting
M(n) : time for merging
b : a constant
c : a constant

4 -12

# Binary search

- e.g. 2 4 5 6 7 8 9



search 7: needs 3 comparisons

- time: O(log n)
- The binary search can be used only if the elements are sorted and stored in an array.

4 -13

# Algorithm binary-search

Input: A sorted sequence of n elements stored in an array.

Output: The position of x (to be searched).

Step 1: If only one element remains in the array, solve it directly.

Step 2: Compare x with the middle element of the array.

Step 2.1: If x = middle element, then output it and stop.

Step 2.2: If x < middle element, then recursively solve the problem with x and the left half array.

Step 2.3: If x > middle element, then recursively solve the problem with x and the right half array.

4 -14

# Algorithm BinSearch(a, low, high, x)

```
// a[]: sorted sequence in nondecreasing order
// low, high: the bounds for searching in a []
// x: the element to be searched
// If x = a[j], for some j, then return j else return –1
 if (low > high) then return –1        // invalid range
 if (low = high) then                  // if small P
   if (x == a[i]) then return i
   else return -1
 else          // divide P into two smaller subproblems
   mid = (low + high) / 2
   if (x == a[mid]) then return mid
   else if (x < a[mid]) then
       return BinSearch(a, low, mid-1, x)
   else return BinSearch(a, mid+1, high, x)
```

4 -15

# Quick Sort

- Small instance has n <= 1. Every small instance is a sorted instance.
- To sort a large instance, select a pivot element from out of the n elements.
- Partition the n elements into 3 groups left, middle and right.
- The middle group contains only the pivot element.
- All elements in the left group are <= pivot.
- All elements in the right group are >= pivot.
- Sort left and right groups recursively.
- Answer is sorted left group, followed by middle group followed by sorted right group.

# Example

| 6 | 2 | 8 | 5 | 11 | 10 | 4 | 1 | 9 | 7 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|

Use 6 as the pivot.

| 2 | 5 | 4 | 1 | 3 | 6 | 7 | 9 | 10 | 11 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|

Sort left and right groups recursively.

# Choice Of Pivot

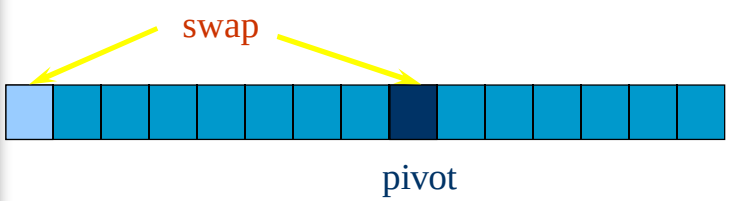
- Pivot is leftmost element in list that is to be sorted.
  - When sorting a[6:20], use a[6] as the pivot.
  - Text implementation does this.
- Randomly select one of the elements to be sorted as the pivot.
  - When sorting a[6:20], generate a random number r in the range [6, 20]. Use a[r] as the pivot.

## Choice Of Pivot

- Median-of-Three rule. From the leftmost, middle, and rightmost elements of the list to be sorted, select the one with median key as the pivot.
  - When sorting a[6:20], examine a[6], a[13] ((6+20)/2), and a[20]. Select the element with median (i.e., middle) key.
  - If a[6].key = 30, a[13].key = 2, and a[20].key = 10, a[20] becomes the pivot.
  - If a[6].key = 3, a[13].key = 2, and a[20].key = 10, a[6] becomes the pivot.

## Choice Of Pivot

- If a[6].key = 30, a[13].key = 25, and a[20].key = 10, a[13] becomes the pivot.
- When the pivot is picked at random or when the median-of-three rule is used, we can use the quick sort code of the text provided we first swap the leftmost element and the chosen pivot.



# Partitioning Into Three Groups

- Sort a = [6, 2, 8, 5, 11, 10, 4, 1, 9, 7, 3].
- Leftmost element (6) is the pivot.
- When another array b is available:
  - Scan a from left to right (omit the pivot in this scan), placing elements <= pivot at the left end of b and the remaining elements at the right end of b.
  - The pivot is placed at the remaining position of the b.

# Quicksort

- Sort into nondecreasing order

```
[26   5  37   1  61  11  59  15  48  19]
[26   5  19   1  61  11  59  15  48  37]
[26   5  19   1  15  11  59  61  48  37]
[11   5  19   1  15] 26 [59  61  48  37]
[11   5   1  19  15] 26 [59  61  48  37]
[ 1   5] 11 [19  15] 26 [59  61  48  37]
  1   5  11  15  19  26 [59  61  48  37]
  1   5  11  15  19  26 [59  37  48  61]
  1   5  11  15  19  26 [48  37] 59 [61]
  1   5  11  15  19  26  37  48  59  61
```

4 -22

# Algorithm Quicksort

Input: A set S of n elements.

Output: The sorted sequence of the inputs in nondecreasing order.

Step 1: If $|S| \leq 2$, solve it directly.

Step 2: (Partition step) Use a pivot to scan all elements in S. Put the smaller elements in $S_1$, and the larger elements in $S_2$.

Step 3: Recursively solve $S_1$ and $S_2$.

4 -23

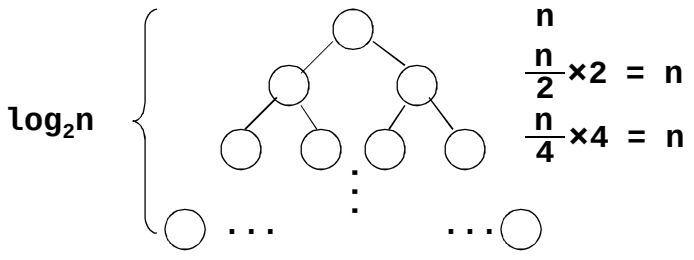
# Time complexity of Quicksort

- time in the worst case:

  $(n-1)+(n-2)+...+1 = n(n-1)/2 = O(n^2)$

- time in the best case:

  In each partition, the problem is always divided into two subproblems with almost equal size.




4 -24

## Time complexity of the best case

- T(n): time required for sorting n elements
- T(n)≦ cn+2T(n/2), for some constant c.

  ≦ cn+2(c . n/2 + 2T(n/4))

  ≦ 2cn + 4T(n/4)

  …

  ≦ $cnlog_2n + nT(1) = O(nlogn)$

4 -25

## The Greedy Method

3 -26

# The knapsack problem

- n objects, each with a weight $w_i > 0$
  a profit $p_i > 0$
  capacity of knapsack: M

Maximize $\sum_{1 \le i \le n} p_i x_i$

Subject to $\sum_{1 \le i \le n} w_i x_i \le M$

$0 \le x_i \le 1, 1 \le i \le n$

3 -27

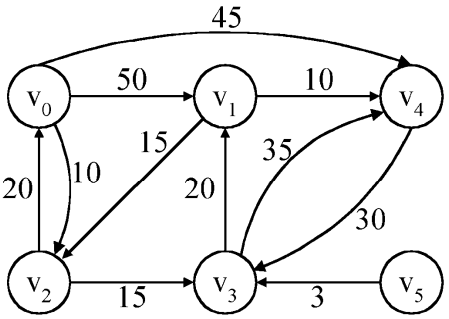
# The knapsack algorithm

- The greedy algorithm:
  Step 1: Sort $p_i/w_i$ into nonincreasing order.
  Step 2: Put the objects into the knapsack according to the sorted sequence as possible as we can.
- e. g.
  n = 3, M = 20, $(p_1, p_2, p_3) = (25, 24, 15)$
  $(w_1, w_2, w_3) = (18, 15, 10)$
  Sol: $p_1/w_1 = 25/18 = 1.39$
  $p_2/w_2 = 24/15 = 1.6$
  $p_3/w_3 = 15/10 = 1.5$
  Optimal solution: $x_1 = 0, x_2 = 1, x_3 = 1/2$

3 -28

# The single-source shortest path problem

- shortest paths from $v_0$ to all destinations




(a)

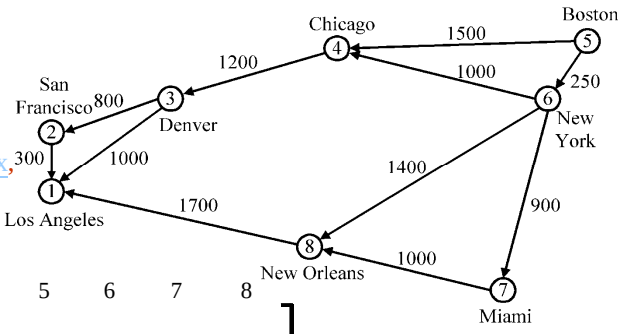
| | Path | Length |
|---|---|---|
| 1) | $v_0v_2$ | 10 |
| 2) | $v_0v_2v_3$ | 25 |
| 3) | $v_0v_2v_3v_1$ | 45 |
| 4) | $v_0v_4$ | 45 |

(b)

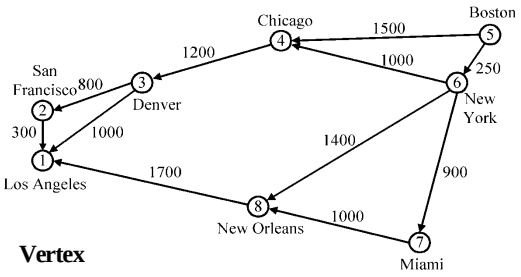


# Dijkstra's algorithm

In the cost adjacency matrix, all entries not shown are $+\infty$.




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | | | | | | | |
| 2 | 300 | 0 | | | | | | |
| 3 | 1000 | 800 | 0 | | | | | |
| 4 | | | 1200 | 0 | | | | |
| 5 | | | | 1500 | 0 | 250 | | |
| 6 | | | | 1000 | | 0 | 900 | 1400 |
| 7 | | | | | | | 0 | 1000 |
| 8 | 1700 | | | | | | | 0 |




| Iteration | S | Vertex Selected | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) |
|---|---|---|---|---|---|---|---|---|---|---|
| Initial | | ---- | | | | | | | | |
| 1 | 5 | 6 | +∞ | +∞ | +∞ | 1500 | 0 | 250 | +∞ | +∞ |
| 2 | 5,6 | 7 | +∞ | +∞ | +∞ | 1250 | 0 | 250 | 1150 | 1650 |
| 3 | 5,6,7 | 4 | +∞ | +∞ | +∞ | 1250 | 0 | 250 | 1150 | 1650 |
| 4 | 5,6,7,4 | 8 | +∞ | +∞ | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 5 | 5,6,7,4,8 | 3 | 3350 | +∞ | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 6 | 5,6,7,4,8,3 | 2 | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| | 5,6,7,4,8,3,2 | | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |

- Time complexity : $O(n^2)$, n = |V|.

3 -31

# Data Structure for SSAD

```
#define MAX_VERTICES 6
int cost[][MAX_VERTICES]=
{};
int distance[MAX_VERTICES];
short int found{MAX_VERTICES];
int n = MAX_VERTICES;
```

adjacency matrix

32

# shortest()

```
Void shortestpath(int v, int cost[][MAX_VERTICES], int dist [], int n, int found[])
{
    int i,u,w;
    for (i=0;i<n;i++) {
            found[i]=FALSE;
            dist [i] = cost[v][i];
    }
    found[v]=TRUE;
    dist [v]=0;
    for(i=0;i<n-2;i++){
            u=choose(dist,n,found);
            found[u]=TRUE;
            for(w=0;w<n;w++)
                    if(dist [u]+cost[u][w] < dist [w])
                        dist [w] = dist [u]+cost[u][w];
    }
}
```

$O(n^2)$

33

## Dynamic programming algorithms for all-pairs shortest path

- We will study a new technique—dynamic programming algorithms (typically for optimization problems)
- **Ideas:**
  - Characterize the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Compute the value of an optimal solution in a bottom-up fashion (using matrix to compute)
  - Backtracking to construct an optimal solution from computed information.

34

# Floyd-Warshall algorithm for shortest path:

- Use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph G=(V,E).
- The resulting algorithm, known as the Floyd-Warshall algorithm, runs in $O(V^3)$ time.
  - negative-weight edges may be present,
  - but we shall assume that there are no negative-weight cycles.

35

# The structure of a shortest path:

- We use a different characterization of the structure of a shortest path than we used in the matrix-multiplication-based all-pairs algorithms.
- The algorithm considers the "intermediate" vertices of a shortest path, where an intermediate vertex of a simple path $p=\langle v_1,v_2,\ldots,v_l\rangle$ is any vertex in p other than $v_1$ or $v_l$, that is, any vertex in the set $\{v_2,v_3,\ldots,v_{l-1}\}$

36

# Continue:

- Let the vertices of G be V={1,2,…,n}, and consider a subset {1,2,…,k} of vertices for some k.
- For any pair of vertices i,j ∈ V, consider all paths from i to j whose intermediate vertices are all drawn from {1,2,…,k},and let p be a minimum-weight path from among them.
- The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set {1,2,…,k-1}.

37

# Relationship:

- The relationship depends on whether or not k is an intermediate vertex of path p.
- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1,2,…,k-1}. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set {1,2,…,k-1} is also a shortest path from i to j with all intermediate vertices in the set {1,2,…,k}.
- If k is an intermediate vertex of path p,then we break p down into $i \xrightarrow{p1} k \xrightarrow{p2} j$ as shown Figure 2.p1 is a shortest path from i to k with all intermediate vertices in the set {1,2,…,k-1}, so as p2.

38

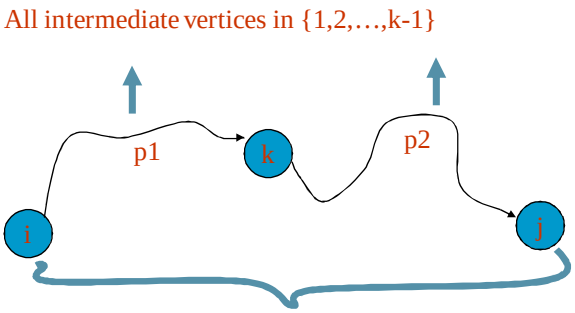



Figure 2. Path p is a shortest path from vertex i to vertex j,and k is the highest-numbered intermediate vertex of p. Path p1, the portion of path p from vertex i to vertex k,has all intermediate vertices in the set {1,2,…,k-1}.The same holds for path p2 from vertex k to vertex j.

39

# A recursive solution to the all-pairs shortest paths problem:

- Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set {1,2,…,k}. A recursive definition is given by
- $$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$
- The matrix $D^{(n)}=(d_{ij}^{(n)})$ gives the final answer-$d_{ij}^{(n)}=\delta(i,j)$ for all $i,j \in V$-because all intermediate vertices are in the set {1,2,…,n}.

40

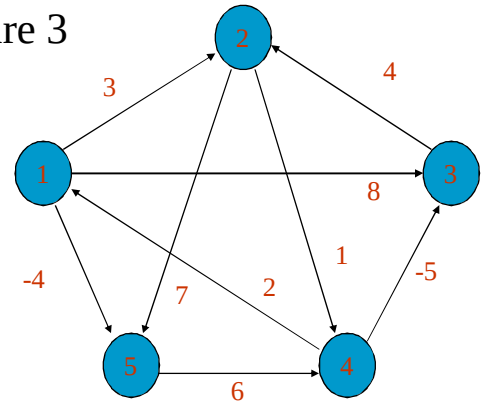
# Computing the shortest-path weights bottom up:

- FLOYD-WARSHALL(W)
- n ← rows[W]
- $D^{(0)}$ ← W
- for k ← 1 to n
- do for i ← 1 to n
- do for j ← 1 to n
- $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
- return $D^{(n)}$

41

# Example:

Figure 3




42

$$D(0)=\begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \prod(0)=\begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D(1)=\begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \prod(1)=\begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

43

$$D(2)=\begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \prod(2)=\begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D(3)=\begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \prod(3)=\begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

44

$$D(4)=\begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \prod(4)=\begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

$$D(5)=\begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \prod(5)=\begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

45