



Techniques Based on Recursion

Lecturer: Kai Wu

Email: kwu@xidian.edu.cn



What's Recursion?

- ***Recursion*** is the process of dividing the problem into one or more *subproblems*, which are identical in structure to the original problem and then combining the solutions to these subproblems to obtain the solution to the original problem.



What's Recursion?

- There are three special cases of the recursion technique:
 - **Induction:** the idea of induction in mathematical proofs is carried over to the design of efficient algorithms
 - **Nonoverlapping subproblems:** divide and conquer
 - **Overlapping subproblems:** dynamic programming, trading space for time



Induction

Given a problem with parameter n , designing an algorithm by induction is based on the fact that if we know how to solve the problem when presented with a parameter less than n , called *the induction hypothesis*, then our task reduces to extending that solution to include those instances with parameter n .



Radix Sort

- Let $L = \{a_1, a_2, \dots, a_n\}$ be a list of n numbers each consisting of exactly k digits. That is, each number is of the form $d_k d_{k-1} \dots d_1$, where each d_i is a digit between 0 and 9.
- In this problem, instead of applying induction on n , the number of objects, we use induction on k , the size of each integer.



Radix Sort

- If the numbers are first distributed into the lists by their ***least significant digit***, then a very efficient algorithm results.
- Suppose that the numbers are sorted lexicographically according to their least $k-1$ digits, i.e., digits $d_{k-1}, d_{k-2}, \dots, d_1$.
- After sorting them on their k th digits, they will eventually be sorted.



Radix Sort

- First, distribute the numbers into 10 lists L_0, L_1, \dots, L_9 according to digit d_1 so that those numbers with $d_1=0$ constitute list L_0 , those with $d_1=1$ constitute list L_1 and so on.
- Next, the lists are coalesced in the order L_0, L_1, \dots, L_9 .
- Then, they are distributed into 10 lists according to digit d_2 , coalesced in order, and so on.
- After distributing them according to d_k and collecting them in order, all numbers will be sorted.



Radix Sort

Example: Sort A nondecreasingly.

$A[1\dots 5] = 7467 \quad 3275 \quad 6792 \quad 9134 \quad 1239$



Radix Sort

- **Input:** A linked list of numbers $L = \{a_1, a_2, \dots, a_n\}$ and k , the number of digits.
- **Output:** L sorted in nondecreasing order.

```
1. for  $j \leftarrow 1$  to  $k$ 
2.   Prepare 10 empty lists  $L_0, L_1, \dots, L_9$ ;
3.   while  $L$  is not empty
4.      $a \leftarrow$  next element in  $L$ ;
5.     Delete  $a$  from  $L$ ;
6.      $i \leftarrow j$ th digit in  $a$ ;
7.     Append  $a$  to list  $L_i$ ;
8.   end while;
9.    $L \leftarrow L_0$ ;
10.  for  $i \leftarrow 1$  to 9
11.     $L \leftarrow L, L_i$  //Append list  $L_i$  to  $L$ 
12.  end for;
13. end for;
14. return  $L$ ;
```



Radix Sort

- **What's the performance of the algorithm Radix Sort?**
 - ✓ Time Complexity?
 - ✓ Space Complexity?



Radix Sort

- Time Complexity: $\Theta(n)$
- Space Complexity: $\Theta(n)$



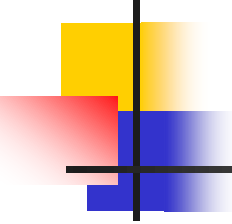
Generating Permutations

- Generating all permutations of the numbers 1, 2, ..., n .
- Based on the assumption that if we can generate all the permutations of $n-1$ numbers, then we can get algorithms for generating all the permutations of n numbers.



Generating Permutations

- Generate all the permutations of the numbers $2, 3, \dots, n$ and add the number 1 to the beginning of each permutation.
- Generate all permutations of the numbers $1, 3, 4, \dots, n$ and add the number 2 to the beginning of each permutation.
- Repeat this procedure until finally the permutations of $1, 2, \dots, n-1$ are generated and the number n is added at the beginning of each permutation.

- 
- **Input:** A positive integer n ;
 - **Output:** All permutations of the numbers $1, 2, \dots, n$;
-

- 1. for $j \leftarrow 1$ to n
- 2. $P[j] \leftarrow j$;
- 3. end for;
- 4. $perm(1)$;

- $perm(m)$
- 1. if $m = n$ then output $P[1 \dots n]$
- 2. else
- 3. for $j \leftarrow m$ to n
- 4. interchange $P[j]$ and $P[m]$;
- //Add one number at the beginning of the permutation
- 5. $perm(m+1)$;
- //Generate permutations for the left numbers
- 6. interchange $P[j]$ and $P[m]$;
- 7. end for;
- 8. end if;



Generating Permutations

- **What's the performance of the algorithm Generating Permutations?**
 - ✓ Time Complexity?
 - ✓ Space Complexity?



Generating Permutations

- Time Complexity: $\Theta(n!)$
- Space Complexity: $\Theta(n)$



Finding the Majority Element

- Let $A[1 \dots n]$ be a sequence of integers. An integer a in A is called the majority if it appears **more than** $\lfloor n/2 \rfloor$ times in A .
- **For example:**
 - ✓ Sequence 1, 3, 2, 3, 3, 4, 3: **3** is the majority element since 3 appears 4 times which is more than $\lfloor n/2 \rfloor$
 - ✓ Sequence 1, 3, 2, 3, 3, 4: **3** is **not** the majority element since 3 appears three times which is equal to $\lfloor n/2 \rfloor$, but not more than $\lfloor n/2 \rfloor$



Finding the Majority Element

- If two **different** elements in the original sequence are **removed**, then the majority in the original sequence remains the majority in the new sequence.
- The above observation suggests the following procedure for finding an element that is candidate for being the majority.

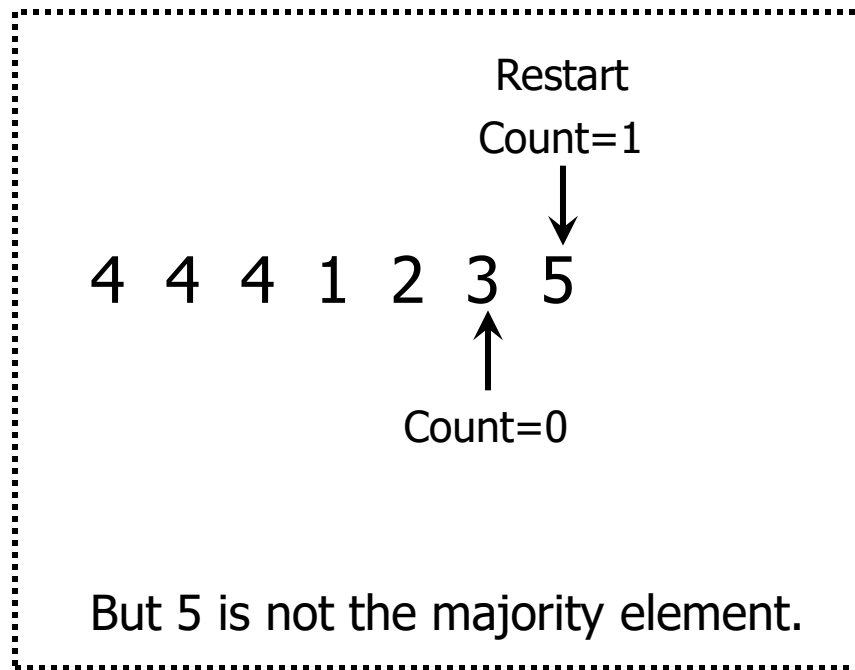


Finding the Majority Element

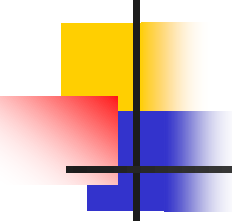
- Let $x=A[1]$ and set a counter to 1.
- Starting from $A[2]$, scan the elements one by one **increasing** the counter by one if the current element is equal to x and **decreasing** the counter by one if the current element is not equal to x .
- If all the elements have been scanned and the counter is greater than zero, then return x as the **candidate**.
- If the counter becomes 0 when comparing x with $A[j]$, $1 < j < n$, then call procedure candidate recursively on the elements $A[j+1 \dots n]$.

Finding the Majority Element

- Why just return x as a candidate?



- **Input:** An array $A[1 \dots n]$ of n elements;
- **Output:** The majority element if it exists; otherwise none;

- 
-
- 1. $x \leftarrow \text{candidate}(1)$;
 - 2. $\text{count} \leftarrow 0$;
 - 3. for $j \leftarrow 1$ to n
 - 4. if $A[j] = x$ then $\text{count} \leftarrow \text{count} + 1$;
 - 5. end for;
 - 6. if $\text{count} > \lfloor n/2 \rfloor$ then return x ;
 - 7. else return none;

- $\text{candidate}(m)$
- 1. $j \leftarrow m$; $x \leftarrow A[m]$; $\text{count} \leftarrow 1$;
- 2. while $j < n$ and $\text{count} > 0$
- 3. $j \leftarrow j + 1$;
- 4. if $A[j] = x$ then $\text{count} \leftarrow \text{count} + 1$;
- 5. else $\text{count} \leftarrow \text{count} - 1$;
- 6. end while;
- 7. if $j = n$ then return x ;
- 8. else return $\text{candidate}(j + 1)$;



Divide and Conquer

- A divide-and-conquer algorithm divides the problem instance into a number of subinstances (in most cases 2), recursively solves each subinstance separately, and then combines the solutions to the subinstances to obtain the solution to the original problem instance.



Divide and Conquer

- Consider the problem of finding both the minimum and maximum in an array of integers $A[1 \dots n]$ and assume for simplicity that n is a power of 2.
- Divide the input array into two halves $A[1 \dots n/2]$ and $A[(n/2)+1 \dots n]$, find the minimum and maximum in each half and return the minimum of the two minima and the maximum of the two maxima.



Divide and Conquer

- **Input:** An array $A[1...n]$ of n integers, where n is a power of 2;
- **Output:** (x, y) : the minimum and maximum integers in A ;
- 1. $minmax(1, n)$;
- $minmax(low, high)$
- 1. if $high - low = 1$ then
- 2. if $A[low] < A[high]$ then return $(A[low], A[high])$;
- 3. else return $(A[high], A[low])$;
- 4. end if;
- 5. else
- 6. $mid \leftarrow \lfloor (low + high) / 2 \rfloor$;
- 7. $(x_1, y_1) \leftarrow minmax(low, mid)$;
- 8. $(x_2, y_2) \leftarrow minmax(mid + 1, high)$;
- 9. $x \leftarrow \min\{x_1, x_2\}$;
- 10. $y \leftarrow \max\{y_1, y_2\}$;
- 11. return (x, y) ;
- 12. end if;



Divide and Conquer

- How many comparisons does this algorithm need?



Divide and Conquer

- Given an array $A[1 \dots n]$ of n elements, where n is a power of 2, it is possible to find both the minimum and maximum of the elements in A using only $(3n/2) - 2$ element comparisons.



The Divide and Conquer Paradigm

- **The divide step:** the input is partitioned into $p \geq 1$ parts, each of size strictly less than n .
- **The conquer step:** performing p recursive call(s) if the problem size is greater than some predefined threshold n_0 .
- **The combine step:** the solutions to the p recursive call(s) are combined to obtain the desired output.



The Divide and Conquer Paradigm

- (1) If the size of the instance I is “small”, then solve the problem using a straightforward method and return the answer. Otherwise, continue to the next step;
- (2) Divide the instance I into p subinstances I_1, I_2, \dots, I_p of approximately the same size;
- (3) Recursively call the algorithm on each subinstance I_j , $1 \leq j \leq p$, to obtain p partial solutions;
- (4) Combine the results of the p partial solutions to obtain the solution to the original instance I . Return the solution of instance I .



Selection: Finding the Median and the k th Smallest Element

- The media of a sequence of n sorted numbers $A[1...n]$ is the “middle” element.
- If n is odd, then the middle element is the $(n+1)/2$ th element in the sequence.
- If n is even, then there are two middle elements occurring at positions $n/2$ and $n/2+1$. In this case, we will choose the $n/2$ th smallest element.
- Thus, in both cases, the median is the $\lceil n/2 \rceil$ th smallest element.
- The k th smallest element is a general case.



Selection: Finding the Median and the k th Smallest Element

- If we select an element mm among A , then A can be divided in to 3 parts:

$$\begin{cases} A_1 = \{a \mid a \in A \cap a < mm\} \\ A_2 = \{a \mid a \in A \cap a = mm\} \\ A_3 = \{a \mid a \in A \cap a > mm\} \end{cases}$$



Selection: Finding the Median and the k th Smallest Element

- According to the number elements in A_1, A_2, A_3 , there are following three cases. In each case, where is the k th smallest element?

$$\begin{cases} |A_1| \geq k \\ |A_1| + |A_2| \geq k \\ |A_1| + |A_2| < k \end{cases}$$



Selection: Finding the Median and the k th Smallest Element

- **Input:** An array $A[1 \dots n]$ of n elements and an integer k , $1 \leq k \leq n$;
- **Output:** The k th smallest element in A ;
- 1. *select*($A, 1, n, k$);

Selection: Finding the Median and the k th Smallest Element

- $select(A, low, high, k)$
 - 1. $p \leftarrow high - low + 1$;
 - 2. if $p < 44$ then sort A and return ($A[k]$);
 - 3. Let $q = \lfloor p/5 \rfloor$. Divide A into q groups of 5 elements each.
 - If 5 does not divide p , then discard the remaining elements;
 - 4. Sort each of the q groups individually and extract its media.
 - Let the set of medians be M .
 - 5. $mm \leftarrow select(M, 1, q, \lceil q/2 \rceil)$;
 - 6. Partition $A[low \dots high]$ into three arrays:
 - $A_1 = \{a \mid a < mm\}$, $A_2 = \{a \mid a = mm\}$, $A_3 = \{a \mid a > mm\}$;
 - 7. case
 - $|A_1| \geq k$: return $select(A_1, 1, |A_1|, k)$;
 - $|A_1| + |A_2| \geq k$: return mm ;
 - $|A_1| + |A_2| < k$: return $select(A_3, 1, |A_3|, k - |A_1| - |A_2|)$;
 - 8. end case;



Selection: Finding the Median and the k th Smallest Element

- What is the time complexity of this algorithm?



Selection: Finding the Median and the k th Smallest Element

- The k th smallest element in a set of n elements drawn from a linearly ordered set can be found in $\Theta(n)$ time.



Quicksort

- Let $A[low \dots high]$ be an array of n numbers, and $x = A[low]$.
- We consider the problem of rearranging the elements in A so that all elements less than or equal to x precede x which in turn precedes all elements greater than x .
- After permuting the element in the array, x will be $A[w]$ for some w , $low \leq w \leq high$. The action of rearrangement is also called *splitting* or *partitioning* around x , which is called the *pivot* or *splitting element*.



Quicksort

- We say that an element $A[j]$ is in its proper position or correct position if it is neither smaller than the elements in $A[low \dots j-1]$ nor larger than the elements in $A[j+1 \dots high]$.
- After partitioning an array A using $x \in A$ as a pivot, x will be in its correct position.



Quicksort

- **Input:** An array of elements $A[low...high]$;
- **Output:** (1) A with its elements rearranged, if necessary;
- (2) w , the new position of the splitting element $A[low]$;

- 1. $i \leftarrow low$;
- 2. $x \leftarrow A[low]$;
- 3. for $j \leftarrow low+1$ to $high$
- 4. if $A[j] \leq x$ then
- 5. $i \leftarrow i+1$;
- 6. if $i \neq j$ then interchange $A[i]$ and $A[j]$;
- 7. end if;
- 8. end for;
- 9. interchange $A[low]$ and $A[i]$;
- 10. $w \leftarrow i$;
- 11. return A and w ;



Quicksort

- The number of element comparisons performed by Algorithm SPLIT is exactly $n-1$. Thus, its time complexity is $\Theta(n)$.
- The only extra space used is that needed to hold its local variables. Therefore, the space complexity is $\Theta(1)$.



Quicksort

- **Input:** An array $A[1 \dots n]$ of n elements;
- **Output:** The elements in A sorted in nondecreasing order;
- 1. *quicksort*($A, 1, n$);
- *quicksort*($A, low, high$)
- 1. if $low < high$ then
- 2. SPLIT($A[low \dots high], w$) \\ w is the new position of $A[low]$;
- 3. *quicksort*($A, low, w-1$);
- 4. *quicksort*($A, w+1, high$);
- 5. end if;



Quicksort

- The average number of comparisons performed by Algorithm QUICKSORT to sort an array of n elements is $\Theta(n \log n)$.



Dynamic Programming

- An algorithm that employs the dynamic programming technique is not recursive by itself, but the underlying solution of the problem is usually stated in the form of a recursive function.
- This technique resorts to evaluating the recurrence in a bottom-up manner, saving intermediate results that are used later on to compute the desired solution.
- This technique applies to many **combinatorial optimization problems** to derive efficient algorithms.



Dynamic Programming

- Fibonacci sequence:

$$f_1=1$$

$$f_2=1$$

$$f_3=2$$

$$f_4=3$$

$$f_5=5$$

$$f_6=8$$

$$f_7=13$$

.....

- $f(n)$
- 1. if ($n=1$) or ($n=2$) then return 1;
- 2. else return $f(n-1)+f(n-2)$;

- This algorithm is far from being efficient, as there are many duplicate recursive calls to the procedure.



Dynamic Programming

- $f(n)$
- 1. if ($n=1$) or ($n=2$) then return 1;
- 2. else
- 3. {
- 4. $fn_1=1$;
- 5. $fn_2=1$;
- 6. for $k \leftarrow 3$ to n
- 7. $fn=fn_1+fn_2$;
- 8. $fn_2=fn_1$;
- 9. $fn_1=fn$;
- 10. end for;
- 11. }
- 12. Return fn ;

- Time: $n-2$ additions $\Rightarrow \theta(n)$
- Space: $\theta(1)$



The Longest Common Subsequence Problem

- Given two strings A and B of lengths n and m , respectively, over an alphabet Σ , determine the length of the longest subsequence that is common to both A and B .
- A subsequence of $A = a_1 a_2 \dots a_n$ is a string of the form $a_{i_1} a_{i_2} \dots a_{i_k}$ where each i_j is between 1 and n and $1 \leq i_1 < i_2 < \dots < i_k \leq n$.



The Longest Common Subsequence Problem

- Let $A = a_1 a_2 \dots a_n$ and $B = b_1 b_2 \dots b_m$.
- Let $L[i, j]$ denote the length of a longest common subsequence of $a_1 a_2 \dots a_i$ and $b_1 b_2 \dots b_j$, $0 \leq i \leq n$, $0 \leq j \leq m$. When i or j be 0, it means the corresponding string is empty.
- Naturally, if $i=0$ or $j=0$; the $L[i, j]=0$



The Longest Common Subsequence Problem

Suppose that both i and j are greater than 0. Then

- If $a_i = b_j$, $L[i, j] = L[i-1, j-1] + 1$
- If $a_i \neq b_j$, $L[i, j] = \max\{L[i, j-1], L[i-1, j]\}$

We get the following recurrence for computing the length of the longest common subsequence of A and B :

$$L[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1, j-1] + 1 & \text{if } i > 0, j > 0 \text{ and } a_i = b_j \\ \max\{L[i, j-1], L[i-1, j]\} & \text{if } i > 0, j > 0 \text{ and } a_i \neq b_j \end{cases}$$



The Longest Common Subsequence Problem

- We use an $(n+1) \times (m+1)$ table to compute the values of $L[i, j]$ for each pair of values of i and j , $0 \leq i \leq n$, $0 \leq j \leq m$.
- We only need to fill the table $L[0 \dots n, 0 \dots m]$ row by row using the previous formula.



The Longest Common Subsequence Problem

- Example: $A = \text{zxyxyz}$, $B = \text{xyyzx}$

What is the longest common subsequence of A and B?



The Longest Common Subsequence Problem

- **Input:** Two strings A and B of lengths n and m , respectively, over an alphabet Σ ;
- **Output:** The length of the longest common subsequence of A and B .
- 1. for $i \leftarrow 0$ to n
- 2. $L[i, 0] \leftarrow 0$;
- 3. end for;
- 4. for $j \leftarrow 0$ to m
- 5. $L[0, j] \leftarrow 0$;
- 6. end for;
- 7. for $i \leftarrow 1$ to n
- 8. for $j \leftarrow 1$ to m
- 9. if $a_i = b_j$ then $L[i, j] \leftarrow L[i-1, j-1] + 1$;
- 10. else $L[i, j] \leftarrow \max\{L[i, j-1], L[i-1, j]\}$;
- 11. end if;
- 12. end for;
- 13. end for;
- 14. return $L[n, m]$;



The Longest Common Subsequence Problem

- **What's the performance of this algorithm?**
 - ✓ Time Complexity?
 - ✓ Space Complexity?



The Longest Common Subsequence Problem

- An optimal solution to the longest common subsequence problem can be found in $\Theta(nm)$ time and $\Theta(\min\{m, n\})$ space.



The Dynamic Programming Paradigm

- The idea of saving solutions to subproblems in order to avoid their recomputation is the basis of this powerful method.
- This is usually the case in many combinatorial optimization problems in which the solution can be expressed in the form of a recurrence whose direct solution causes subinstances to be computed more than once.



The Dynamic Programming Paradigm

- An important observation about the working of dynamic programming is that the algorithm computes an optimal solution to every subinstance of the original instance considered by the algorithm.
- This argument illustrates an important principle in algorithm design called the *principle of optimality*. Given an optimal sequence of decisions, each subsequence must be an optimal sequence of decisions by itself.

The All-Pairs Shortest Path Problem

- Let $G=(V, E)$ be a directed graph in which each edge (i, j) has a non-negative length $\ell[i, j]$. If there is no edge from vertex i to vertex j , then $\ell[i, j]=\infty$.
- The problem is to find the distance from each vertex to all other vertices, where the distance from vertex x to vertex y is the length of a shortest path from x to y .
- For simplicity, we will assume that $V=\{1, 2, \dots, n\}$. Let i and j be two different vertices in V . Define $d_{i,j}^k$ to be the length of a shortest path from i to j that does not pass through any vertex in $\{k+1, k+2, \dots, n\}$.

The All-Pairs Shortest Path Problem



- $d_{i,j}^0 = l[i, j]$
- $d_{i,j}^1$ is the length of a shortest path from i to j that does not pass through any vertex except possibly vertex 1
- $d_{i,j}^2$ is the length of a shortest path from i to j that does not pass through any vertex except possibly vertex 1 or vertex 2 or both
- $d_{i,j}^n$ is the length of a shortest path from i to j , i.e. the distance from i to j



The All-Pairs Shortest Path Problem

- We can compute $d_{i,j}^k$ recursively as follows:

$$d_{i,j}^k = \begin{cases} l[i, j] & \text{if } k = 0 \\ \min \{ d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1} \} & \text{if } 1 \leq k \leq n \end{cases}$$

The All-Pairs Shortest Path Problem

- **Floyd Algorithm:** use $n+1$ matrices $D_0, D_1, D_2, \dots, D_n$ of dimension $n \times n$ to compute the lengths of the shortest constrained paths.
- Initially, we set $D_0[i, i] = 0$, $D_0[i, j] = \ell[i, j]$ if $i \neq j$ and (i, j) is an edge in G ; otherwise $D_0[i, j] = \infty$.
- We then make n iterations such that after the k th iteration, $D_k[i, j]$ contains the value of a shortest length path from vertex i to vertex j that does not pass through any vertex numbered higher than k .

The All-Pairs Shortest Path Problem

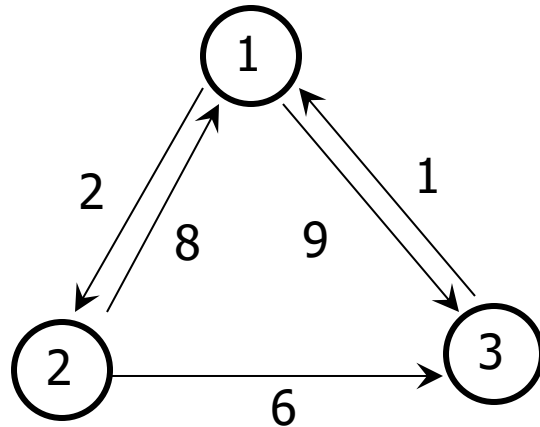


- Thus, in the k th iteration, we compute $D_k[i, j]$ using the formula

$$D_k[i, j] = \min\{D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j]\}$$

The All-Pairs Shortest Path Problem

- Example:



The All-Pairs Shortest Path Problem

- **Input:** An $n \times n$ matrix $\ell[1 \dots n, 1 \dots n]$ such that $\ell[i, j]$ is the length of the edge (i, j) in a directed graph $G = (\{1, 2, \dots, n\}, E)$;
- **Output:** A matrix D with $D[i, j]$ = the distance from i to j ;
- 1. $D \leftarrow \ell$;
- 2. for $k \leftarrow 1$ to n
- 3. for $i \leftarrow 1$ to n
- 4. for $j \leftarrow 1$ to n
- 5. $D[i, j] = \min\{D[i, j], D[i, k] + D[k, j]\}$;
- 6. end for;
- 7. end for;
- 8. end for;

The All-Pairs Shortest Path Problem



- What is the time and space complexity of the FLOYD algorithm?



The All-Pairs Shortest Path Problem

- The running time of FLOYD algorithm is $\Theta(n^3)$ and its space complexity is $\Theta(n^2)$.



The Knapsack Problem

- Let $U = \{u_1, u_2, \dots, u_n\}$ be a set of n items to be packed in a knapsack of size C . for $1 \leq j \leq n$, let s_j and v_j be the size and value of the j th item, respectively, where C and $s_j, v_j, 1 \leq j \leq n$, are all positive integers.
- The objective is to fill the knapsack with some items for U whose total size is at most C and such that their total value is maximum. Assume without loss of generality that the size of each item does not exceed C .



The Knapsack Problem

- More formally, given U of n items, we want to find a subset $S \subseteq U$ such that

$$\sum_{u_i \in S} v_i$$

is maximized subject to the constraint

$$\sum_{u_i \in S} s_i \leq C$$

- This version of the knapsack problem is sometimes referred to in the literature as the 0/1 knapsack problem. This is because the knapsack cannot contain more than one item of the same type.



The Knapsack Problem

- Let $V[i, j]$ denote the value obtained by filling a knapsack of size j with items taken from the first i items $\{u_1, u_2, \dots, u_i\}$ in an optimal way. Here the range of i is from 0 to n and the range of j is from 0 to C . Thus, what we seek is the value $V[n, C]$.
- Obviously, $V[0, j]$ is 0 for all values of j , as there is nothing in the knapsack. On the other hand, $V[i, 0]$ is 0 for all values of i since nothing can be put in a knapsack of size 0.



The Knapsack Problem

- $V[i, j]$, where $i > 0$ and $j > 0$, is the maximum of the following two quantities:
 - ✓ $V[i-1, j]$: The maximum value obtained by filling a knapsack of size j with items taken from $\{u_1, u_2, \dots, u_{i-1}\}$ only in an optimal way.
 - ✓ $V[i-1, j-s_i] + v_i$: The maximum value obtained by filling a knapsack of size $j-s_i$ with items taken from $\{u_1, u_2, \dots, u_{i-1}\}$ in an optimal way plus the value of item u_i . This case applies only if $j \geq s_i$ and it amounts to adding item u_i to the knapsack.



The Knapsack Problem

- Then, we got the following recurrence for finding the value in an optimal packing:

$$V[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1, j] & \text{if } j < s_i \\ \max \{V[i-1, j], V[i-1, j-s_i] + v_i\} & \text{if } j \geq s_i \end{cases}$$

- Using dynamic programming to solve this integer programming problem is now straightforward. We use an $(n+1) \times (C+1)$ table to evaluate the values of $V[i, j]$. We only need to fill the table $V[0 \dots n, 0 \dots C]$ row by row using the above formula.



The Knapsack Problem

- **Example:**

$$C=9$$

$$U=\{u_1, u_2, u_3, u_4\}$$

$$S_i=2, 3, 4, 5$$

$$V_i=3, 4, 5, 7$$



The Knapsack Problem

- **Input:** A set of items $U = \{u_1, u_2, \dots, u_n\}$ with sizes s_1, s_2, \dots, s_n and values v_1, v_2, \dots, v_n and a knapsack capacity C .
- **Output:** The maximum value of the function $\sum_{u_i \in S} v_i$ subject to $\sum_{u_i \in S} s_i \leq C$ for some subset of items $S \subseteq U$.
- 1. for $i \leftarrow 0$ to n
- 2. $V[i, 0] \leftarrow 0$;
- 3. end for;
- 4. for $j \leftarrow 0$ to C
- 5. $V[0, j] \leftarrow 0$;
- 6. end for;
- 7. for $i \leftarrow 1$ to n
- 8. for $j \leftarrow 1$ to C
- 9. $V[i, j] \leftarrow V[i-1, j]$;
- 10. if $s_i \leq j$ then $V[i, j] \leftarrow \max\{V[i, j], V[i-1, j-s_i] + v_i\}$;
- 11. end for;
- 12. end for;
- 13. return $V[n, C]$;



The Knapsack Problem

- What is the time and space complexity of this algorithm?



The Knapsack Problem

- An optimal solution to the Knapsack problem can be found in $\Theta(nC)$ time and $\Theta(C)$ space.