CS-203 – WEEK 03 – BRUTE FORCE AND EXHAUSTIVE SEARCH

GIUSEPPE TURINI

Kettering
UNIVERSITY

TABLE OF CONTENTS

STUDY GUIDE

Study Material:

- These slides.
- Your notes.
- * "Introduction to the Design and Analysis of Algorithms (3rd Ed.)", chap. 3, pp. 97-130.
- "Introduction to the Design and Analysis of Algorithms (3rd Ed.)", chap. 4, pp. 144-149.

Practice Exercises:

- Exercises from *: 3.1.1-5, 3.1.8-14, 3.2.2-5, 3.2.8, 3.3.1, 3.3.3-5, 3.3.7, 3.4.1-3, 3.4.6-9, 3.5.1-4, 3.5.6-7, 3.5.9-10.

Additional Resources:

- VisuAlgo

INTRODUCTION TO BRUTE FORCE AND EXHAUSTIVE SEARCH

The first 2 algorithm designs that we discuss are: brute force, and exhaustive search.

Brute Force: A straightforward approach to solving a problem, usually directly based on the problem statement and definition of the concepts involved.

For example: computing a^n by multiplying the base (a) by itself as many times (n-1) as required by the exponent (n).

Exhaustive Search: To test each possible "value" sequentially in order to determine if it is the correct solution. The implicit assumption here is that we can check is a value is the correct solution, but we do not know how to "build" the solution step-by-step.

It is a special case of brute-force approach.

For example: finding a^n by testing each integer (starting from 0) until we find a value that is the correct solution (e.g., a value v for which $v^{1/n} = a$).

BRUTE FORCE

Brute Force: A straightforward approach to solving a problem, usually directly based on the problem statement and definition of the concepts involved.

For example: computing a^n by multiplying the base (a) by itself as many times (n-1) as required by the exponent (n).

The brute-force approach is an important algorithm design for these reasons:

- The brute-force strategy is applicable to a wide variety of problems.
- For some problems the brute-force approach yields algorithms of practical value.
- A brute-force algorithm can be reasonable/efficient if the input size is small.
- A brute-force algorithm can serve as a benchmark to judge other algorithms.

SELECTION SORT: ALGORITHM

Problem: Sort a list of n orderable items, rearranging them in non-decreasing order.

Selection Sort

Step 1: Scan the entire list to find its smallest item.

Step 2: Put the smallest item in its final spot in the sorted list.

Step 3: Repeat Step 1 but scan begins at the next spot.

Step 4: The algorithm stops when scan starts at last item.

| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

Figure 1. Selection Sort of list { 89, 45, 68, 90, 29, 34, 17 }. Each row is 1 pass of the algorithm. Separator splits the list between its sorted/unsorted partitions.

SELECTION SORT: IMPLEMENTATION

This is a Java implementation of Selection Sort.

```
public static void SelectionSort( int[] A ) {
   // Temp variables.
   int min; // Index of current partition minimum.
   int temp; // Buffer variable to perform swap.
   // Loop to sort 1 item at time (except last).
   for( int i = 0; i < A.length - 1; i++ ) {
      min = i; // Init/reset index of current partition minimum.
      // Scan array partition [i+1,n-1] to find partition minimum.
      for( int j = i+1; j < A.length; j++ ) {
         // Compare current partition minimum against current partition scan item.
         if( A[j] < A[min] ) { min = j; }
      }
      // Put partition minimum in its final sorted spot (swap A[i] and A[min]).
      temp = A[i];
      A[i] = A[min];
      A[min] = temp;
   }
}
```

SELECTION SORT: ANALYSIS

We need n-1 scans to sort a list (A) with n items (assume scans are numbered from 0 to n-2).
These scans have decreasing lengths (scan 0 is the longest, scan n-2 is the shortest).
On the i^{th} scan through the list, Selection Sort:

- Searches for the smallest item in the last n-i list items.
- Then, and swaps the smallest item with list item at index i (A_i).

$$\underbrace{A_0 \leq A_1 \leq \cdots \leq A_{i-1}}_{\text{items in final position}} \quad | \quad \underbrace{A_i, \cdots, A_{min}, \cdots, A_{n-1}}_{\text{last } n-i \text{ items}}$$

The input size is the item-size of the list, and the basic operation is the comparison.
The basic operation count C(n) depends only on the input size.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [\,(n-1) - (i+1) + 1\,] = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)\,n}{2} \in \Theta(n^2)$$

BUBBLE SORT: ALGORITHM

Problem: Sort a list of n orderable items, rearranging them in non-decreasing order.

Bubble Sort v1

Step 1: Scan the entire list comparing adjacent items.

Step 2: Swap adjacent items if they are out of order.

Step 3: Repeat Step 1 but scan begins at the next spot.

Step 4: The algorithm stops when scan starts at last item.

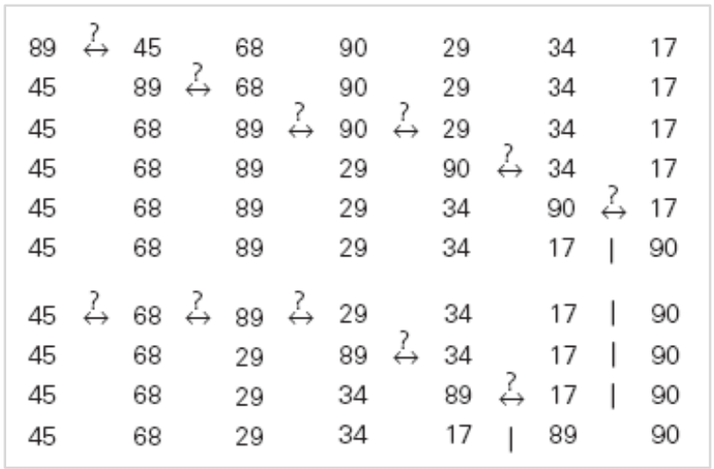


Figure 2. Two passes of Bubble Sort of list { 89, 45, 68, 90, 29, 34, 17 }. Each item comparison/swap is marked by a double-arrow. Separator splits the list between its sorted/unsorted partition

BUBBLE SORT: IMPLEMENTATION

This is a Java implementation of Bubble Sort.

```
public static void BubbleSort1( int[] A ) {
   // Loop to perform n-2 scans.
   for( int i = 0; i < A.length - 1; i++ ) {
      // Bubble up items (partial scan).
      for( int j = 0; j < A.length - 1 - i; j++ ) {
         // Check if adjacent items are out of order.
         if( A[j+1] < A[j] ) {
            // Swap adjacent items.
            int temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
         }
      }
   }
}
```

BUBBLE SORT: ANALYSIS

The input size is the item-size of the list, and the basic operation is the comparison.
The basic operation count C(n) depends only on the input size.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [\,(n-2-i) - 0 + 1\,] = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)\,n}{2} \in \Theta(n^2)$$

Note: The number of swaps S(n), depends on the input.
In the best-case (increasing list) the algorithm performs 0 swaps ($S_{best}(n)$).
In the worst-case (decreasing list) the algorithm does 1 swap per comparison ($S_{worst}(n)$).

$$S_{best}(n) = 0 \in \Theta(1) \qquad S_{worst}(n) = \frac{(n-1)\,n}{2} \in \Theta(n^2)$$

$$S(n) \in \Omega(1) \qquad S(n) \in O(n^2)$$

BUBBLE SORT: OPTIMIZATION

We can improve the original version of Bubble Sort by observing that: if a pass through the list makes no swaps, the list is sorted and we can stop.

This optimized version of Bubble Sort runs faster on some inputs.
So, its average-case efficiency ($C_{avg}(n)$) will probably be slightly better.
However, the optimized Bubble Sort is still in the same efficiency class:

- $C_{opt\text{-}worst}(n)$ is in $\Theta(n^2)$.
- $C_{opt\text{-}avg}(n)$ is in $\Theta(n^2)$.
- $C_{opt}(n)$ is in $O(n^2)$.

Note: This is a common situation with brute force approaches, that is: the original brute force algorithm often results in a strategy that can be easily improved.

BUBBLE SORT: OPTIMIZATION 2

This is a Java implementation of Bubble Sort (optimized version), including commands (highlighted) allowing an early return when a scan makes no swaps (list sorted).

```
public static void BubbleSort2( int[] A ) {
   // Loop to perform n-2 scans.
   for( int i = 0; i < A.length - 1; i++ ) {
      // Bubble up items (partial scan).
      boolean swapMade = false; // Init flag to track swaps made.
      for( int j = 0; j < A.length - 1 - i; j++ ) {
         // Check if adjacent items are out of order.
         if( A[j+1] < A[j] ) {
            // Swap adjacent items.
            int temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
            swapMade = true; // Update flaf to track swaps made.
         }
      }
      // If no swap was made, early return (list sorted).
      if(!swapMade) { return; }
   }
}
```

SEQUENTIAL SEARCH: ALGORITHM AND IMPLEMENTATION

Problem: Search for an item in a list, and return its position-index (if found) or -1.

Sequential Search v1 Scan a list, comparing each item with input key, until either a match is found (success) or the scan ends without a match (failure).

This is a Java implementation of Sequential Search.

```
public static int SequentialSearch1( int[] A, int k ) {
   int i = 0; // Scan index.
   // Sequential list scan: stop at list end or when key is found.
   while( ( i < A.length ) && ( A[i] != k ) ) { i++; }
   // Check if search is successful.
   if( i < A.length ) { return i; }
   else { return -1; }
}
```

SEQUENTIAL SEARCH: ANALYSIS

The input size is the item-size of the list, and the basic operation is the comparison.
The basic operation count C(n) does not depend only on the input size, but also on the list content; so, we need best-case, worst-case, and average-case analyses.

- The **worst-case** happens for lists with no matching item or with a match at last item.

$$C_{worst}(n) = \sum_{i=0}^{n-1} 1 = n \in \Theta(n)$$

- The **best-case** happens for lists with a match at first item.

$$C_{best}(n) = 1 \in \Theta(1)$$

- So, the general count C(n) is in: $C(n) \in O(n)$ and $C(n) \in \Omega(1)$

SEQUENTIAL SEARCH: OPTIMIZATION

We can improve the original version of Sequential Search in different ways.

- We can avoid failed searches by extending the list: by appending the search key to the end of the list. In this way: any search will be successful, we can eliminate the end of the list check, but we need to remember that a match at the end means "key not found".

- A simple improvement can be done for sorted lists. In these cases, the search can be stopped (early return) as soon as a list item greater than the search key is found.

BRUTE-FORCE STRING MATCHING: ALGORITHM

Problem: Given a string T (text) of n characters t_0 ... t_{n-1}, and another string P (pattern) of m characters p_0 ... p_{m-1} ($m \leq n$): find a substring of text T that matches pattern P.

That is, find the index (i) of leftmost character of first matching substring in T.

Brute-Force String Matching

Align the pattern against the first m characters of text T.
Start matching pairs of characters from left to right.
Stop if all m pairs match (success).
When a mismatch is found, right-shift pattern P by 1, and resume the character comparison.

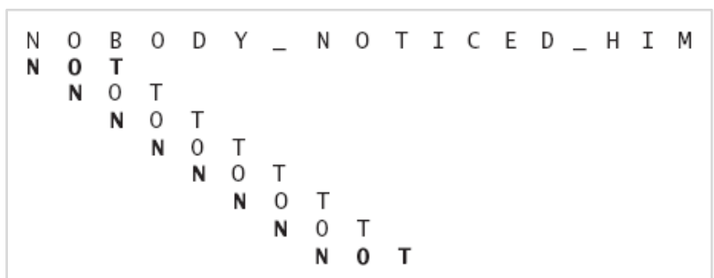


Figure 3. Example of Brute-Force String Matching. The pattern ("NOT") is repeatedly compared with the characters in the text ("NOBODY NOTICED HIM").

BRUTE-FORCE STRING MATCHING: IMPLEMENTATION

This is a Java implementation of Brute-Force String Matching.

```java
public static int BruteForceStringMatching( String T, String P ) {
   // Init text T and pattern P sizes.
   int n = T.length();
   int m = P.length();
   // Scan text T (stop at n-m character).
   for( int i = 0; i <= n - m; i++ ) {
      // Init matching index.
      int j = 0;
      // Start matching pattern P.
      while( ( j < m ) && ( P.charAt(j) == T.charAt(i+j) ) ) { j++; }
      // Check if matching was successful.
      if( j == m ) { return i; }
   }
   // Search failed.
   return -1;
}
```

Note: The last index in text T that can be used as starting point for the pattern matching is n-m (text characters indexed from 0 to n-1).

BRUTE-FORCE STRING MATCHING: ANALYSIS

The input size includes both the text T size (n) and the pattern P size (m).
The basic operation is the character comparison.
The basic operation count C(n,m) does not depend only on the input size (text/pattern sizes), but also on the text/pattern contents; so, we need best-, worst-, and average-case analyses.

- The **worst-case** happens when the algorithm makes all m comparisons before shifting the pattern, for each of the n-m+1 matching attempts.

$$C_{worst}(n,m) = \sum_{i=0}^{n-m} m = m(n-m+1) \in \Theta(n\,m)$$

- The **best-case** happens when pattern is matched at index 0 (with m comparisons).

$$C_{best}(n,m) = m \in \Theta(m)$$

- So, the general count C(n,m) is in: $C(n,m) \in O(n\,m)$ and $C(n,m) \in \Omega(m)$

BRUTE-FORCE CLOSEST-PAIR: ALGORITHM

Problem: Find the closest pair of 2D points in a set of n 2D points (distinct points).

Assume 2D points are specified by their (x, y) Cartesian coordinates, and the distance between 2D points p_i (x_i, y_i) and p_j (x_j, y_j) is the Euclidean distance $d(p_i, p_j)$.

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Brute-Force Closest-Pair

Step 1: Compute the distance of each pair of 2D points.

Step 2: Find the pair with the smallest distance.

To avoid computing the distance between the same pair twice, we consider only pairs of points (p_i, p_j) for which ($i < j$).

BRUTE-FORCE CLOSEST-PAIR: IMPLEMENTATION

This is a Java implementation of Brute-Force Closest-Pair.

```
public static double ClosestPair2D( Point[] P ) {
   // Init closest pair distance with max double value.
   double d = Double.MAX_VALUE;
   // Double loop to setup point pairs (avoiding duplicates).
   for( int i = 0; i < ( P.length - 1 ); i++ ) {
      for( int j = i+1; j < P.length; j++ ) {
         // Update closest pair distance.
         d = Math.min( d, P[i].distance( P[j] ) );
      }
   }
   // Return closest pair distance (note that closest pair is not returned).
   return d;
}
```

BRUTE-FORCE CLOSEST-PAIR: ANALYSIS

The input size is the number of 2D points in input (n).
The basic operation of the algorithm is computing the square root (or the distance function) since it is the most time-consuming operation among the most executed.
The basic operation count C(n) depends only on the input size; so, no need of best-, worst-, and average-case analyses.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i}^{n-1} 1 = \sum_{i=0}^{n-2} (n-2-i+1) = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)\,n}{2} \in \Theta(n^2)$$

BRUTE-FORCE CLOSEST-PAIR: OPTIMIZATION

We can improve the time efficiency of the brute-force closest-pair algorithm by avoiding using the square root (to compute the distance), and instead computing only the square distance, without affecting the correctness of the algorithm. If we do so, the basic operation will be squaring a number, and the basic operation count $C_{opt}(n)$ will be:

$$C_{opt}(n) = \sum_{i=0}^{n-2} \sum_{j=i}^{n-1} 2 = \sum_{i=0}^{n-2} 2\,(n-2-i+1) = \sum_{i=0}^{n-2} 2\,(n-1-i) = (n-1)\,n \in \Theta(n^2)$$

Question: Why the count $C_{opt}(n)$ (optimized alg.) is twice the count $C(n)$ (original alg.)?

Answer: $C_{opt}(n)$ counts the squaring of a number, whereas $C(n)$ counts the square roots performed. So, these two counts cannot be compared directly.

Note: This optimization speeds up the algorithm (by avoiding executing ~n^2 square roots), but it does not improve its asymptotic efficiency class (both versions are in $\Theta(n^2)$).

INTRODUCTION TO COMBINATORIAL OBJECTS

Combinatorics is the mathematics field focused on counting/properties of finite structures.

In particular, enumerative combinatorics is the sub-field of combinatorics focused on the number of ways to form certain patterns (e.g., counting combinations/permutations, etc.).

The most commonly used combinatorial objects are:

- Permutations.
- Combinations.
- Subsets.

What follows is a brief introduction to the main combinatorial objects that are frequently used in designing brute-force and exhaustive-search algorithms.

PERMUTATIONS

Permutations: In mathematics, a permutation is a way of selecting distinct items from a set, such that the order of selection matters.

Selections of a fixed length k of distinct items taken from a given set of size n, are called k-permutations of n and their number P(n,k) is:

$$\mathrm{P}(n,k) = \begin{cases} n!/(n-k)!, & \text{if } k \leq n. \\ 0, & \text{if } k > n. \end{cases}$$

Example:

Given a set of 4 characters:	S = { a, b, c, d }
The 2-permutations of 4 of S are:	{ ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc }
Their number is:	P(4,2) = 4! / (4-2)! = 12

COMBINATIONS

Combinations: In mathematics, a combination is a way of selecting distinct items from a set, such that the order of selection does not matter.

Selections of a fixed length k of distinct items taken from a given set of size n, are called k-combinations of n and their number C(n,k) is:

$$\mathrm{C}(n,k) = \text{binomial coefficient}\binom{n}{k} = \begin{cases} \dfrac{n!}{k!\,(n-k)!}, & \text{if } k \le n. \\ 0, & \text{if } k > n. \end{cases}$$

Example: Given a set of 4 characters: S = { a, b, c, d }
The 2-combinations of 4 of S are: { ab, ac, ad, bc, bd, cd }
Their number is: C(4,2) = 4! / 2! (4-2)! = 6

PERMUTATIONS AND COMBINATIONS

Note: By taking all the k-combinations of a set S with n items and ordering each of these combinations in all possible ways we obtain all the k-permutations of S:

$$\mathrm{C}(n,k) \times \mathrm{P}(k,k) = \mathrm{P}(n,k)$$

$$\Downarrow$$

$$\mathrm{C}(n,k) = \frac{\mathrm{P}(n,k)}{\mathrm{P}(k,k)} = \frac{n!}{k!\,(n-k)!} = \text{binomial coefficient} \binom{n}{k}$$

Example: Given a set of 4 characters: S = { a, b, c, d }
The 2-combinations of 4 of S are: { ab, ac, ad, bc, bd, cd }
The 2-permutations of 4 of S are: { ab, ba, ac, ca, ad, da, bc, cb, bd, db, cd, dc }

SUBSETS AND THE POWER SET

Power Set: In mathematics, the power set P(S) of any set S, is the set of all subsets of S (including the empty set, and S itself).

If a set S has n items, then the number of subsets of S is: $|P(S)| = 2^n$.

Note: To demonstrate that the size of the power set P(S) of a set S (with n items) is 2^n, it is convenient to consider a generic subset S_i of S as represented by a bitmask of size n (where each bit represents the presence/absence of the relative item of S in S_i).

Example: Given a set of 3 characters: $S = \{ a, b, c \}$

The power set P(S) of S is: $P(S) = \{ \emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\} \}$

The size of P(S) is: $|P(S)| = 2^3 = 8$

A subset-bitmask could be: $S_i = 101 = \{a,c\}$

GENERATING PERMUTATIONS

Johnson-Trotter Algorithm

This algorithm generates all the n-permutations of n by associating a swap direction to each element e in a permutation. If the direction of element e points to a smaller adjacent element, then the element e is said to be "mobile".
At each iteration:

- **Step 1:** Select the max mobile element e.
- **Step 2:** Swap e with its adjacent (following direction).
- **Step 3:** Reverse direction of elements greater than e.
- **Step 4:** Stop when there are no mobile elements.

Example: The following 3-permutation of 3 (digits 1, 2, and 3) is an example of an arrow-marked permutation used by the Johnson-Trotter algorithm.
Below, the mobile element swapped at each iteration is highlighted:

$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{3} \Rightarrow \overleftarrow{1}\,\overleftarrow{3}\,\overleftarrow{2} \Rightarrow \overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{2} \Rightarrow \overrightarrow{3}\,\overleftarrow{2}\,\overleftarrow{1} \Rightarrow \overleftarrow{2}\,\overrightarrow{3}\,\overleftarrow{1} \Rightarrow \overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}$$

GENERATING PERMUTATIONS 2

This is a Java implementation of the Johnson-Trotter algorithm.

```
public static void JohnsonTrotterAlgorithm( int n ) {
   // Initialize the first permutation.
   int[] P = new int[ n ]; // Permutation array.
   boolean[] D = new boolean[ n ]; // Direction array (true = left, false = right).
   for( int i = 0; i < n; i++ ) {
      P[i] = i+1; D[i] = true;
   }
   PrintPermutation( P );
   // Generate all the permutation via minimal-change requirement.
   while( PermutationHasMobileElement( P, D ) ) {
      int maxMobileIndex = FindLargestMobileValue( P, D );
      int maxMobileValue = P[maxMobileIndex];
      SwapMobileValue( P, D, maxMobileIndex );
      ReverseDirOfValuesGreaterThanMobile( P, D, maxMobileValue );
      PrintPermutation( P );
   }
}
```

GENERATING PERMUTATIONS 3

Note: The Johnson-Trotter algorithm is designed to generate n-permutations of n; so, if you need k-permutations of n, you have to start with generating all the initial k-combinations of n before applying Johnson-Trotter.

Note: The Johnson-Trotter algorithm is one of the most efficient for generating permutations: it can run in time proportional to the number of permutations $\Theta(n!)$.

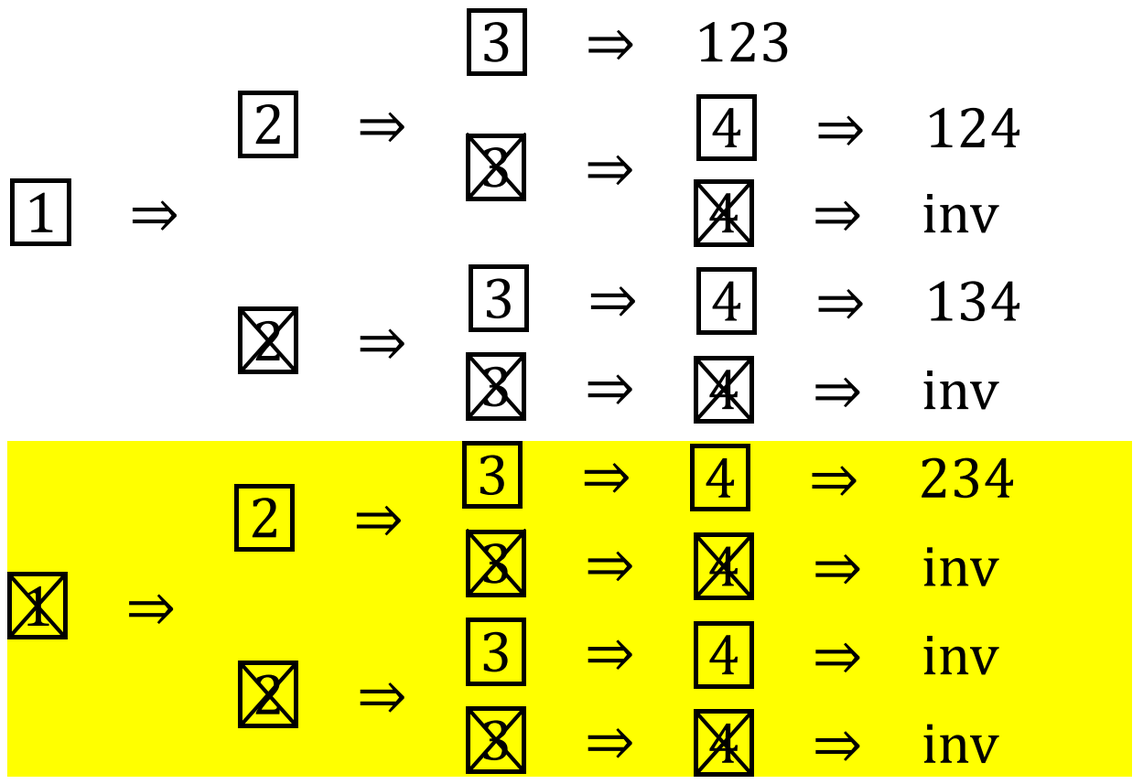
Note: Even if it is listed here, the Johnson-Trotter algorithm is a minimal-change decrease-by-one algorithm (not a brute-force algorithm).

Note: The Johnson-Trotter algorithm is only one of the algorithms to generate permutations (e.g., another commonly used algorithm is the Lexicographic Permute).

GENERATING COMBINATIONS

Pascal's Rule Algorithm Generates all the k-permutations of n by excluding/including 1-by-1 each item of the input set (sequentially).

Example: This is the generation of the 3-combinations of 4 (digits 1, 2, 3, and 4):



GENERATING COMBINATIONS 2

This is a Java implementation of the Pascal's Rule algorithm.

```
private static void GenCombinationsRec(int[] A, int[] C, int k, int iC, int iA) {
   // Check if current combination C is ready to be printed.
   if(indexC == k) { PrintCombination(C); return; }
   // Check if there are no more items in input set.
   if(iA >= A.length) { return; }
   // CASE 1: Include current item.
   C[iC] = A[iA];
   GenCombinationsRec(A, C, k, iC+1, iA+1);
   // CASE 2: Exclude current item.
   GenCombinationsRec(A, C, k, iC, iA+1);
}

public static void GenCombinations(int n, int k) {
   // Initialize item set.
   int[] A = new int[ n ];
   for( int i = 0; i < n; i++ ) { A[i] = i+1; }
   int[] C = new int[k]; // Init current combination.
   // Generate (and print) all k-combinations of n (recursively).
   GenCombinationsRec(A, C, k, 0, 0);
}
```

GENERATING SUBSETS

To generate all 2^n subsets in the power set P(A) of a set $A = \{ a_1, a_2, \dots , a_n \}$, the key idea is that, given an item a_i of A, all subsets of A can be split into 2 groups:

- Subsets of A excluding a_i.
- Subsets of A including a_i.

Each subset of the 2^{nd} group (subsets including a_i) can be obtained by adding a_i to each subset of the 1^{st} group (subsets excluding a_i). So, starting from the empty set (ø), we can generate all other subsets by sequentially adding all other items in set A one by one (see Figure 4).

n	subsets							
0	∅							
1	∅	$\{a_1\}$						
2	∅	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	∅	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

Figure 4. All the subsets of the set $\{a_1, a_2, a_3\}$ generated bottom-up. Each row shows the subsets generated after adding the relative set item, for example: row 2 shows the subsets generated after adding item a_2 to the previously generated subsets.

GENERATING SUBSETS 2

To create all the subsets of A, we do not have to create all power sets of the smaller sets.

Consider the 1-to-1 correspondence between all 2^n subsets of a set $A = \{ a_1, a_2, \ldots, a_n \}$ with n items and all 2^n bitstrings of length n (e.g., $b_1 b_2 \ldots b_n$), for example:

- $b_i = 1$ (i^{th} bit in bitstring) if a_i (i^{th} item in set A) belongs to the subset.
- $b_i = 0$ if a_i does not belong to the subset.

Representing subsets with bitstrings eases the generation of all subsets because we can generate successive binary numbers from 0 to 2^n-1 (padded with leading 0s, when necessary).

Example: For a set $A = \{ a_1, a_2, a_3 \}$ with n = 3 items, we obtain:

bitstring	000	001	010	011	100	101	110	111
subset	$\emptyset$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

GENERATING SUBSETS 3

Note: Although the bitstrings generated by the previous strategy are in lexicographic order, the order of the subsets is not "natural".

Binary Reflected Gray Code Algorithm to Generate Subsets

This algorithm is a minimal-change decrease-by-1 algorithm for generating bitstrings so that every one of them differs from its immediate predecessor by only a single bit.

Example: For a set A = { a_1, a_2, a_3 } with n = 3 items, we obtain:

bitstring	000	001	011	010	110	111	101	100
subset	$\emptyset$	$\{a_3\}$	$\{a_2, a_3\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$	$\{a_1, a_3\}$	$\{a_1\}$

GENERATING SUBSETS 4

This is a Java implementation of the Binary Reflected Gray Code algorithm.

```
public static int[][] BinaryReflectedGrayCodeRec( int n ) {
   // Init temp list of bitstrings (subsets).
   int[][] L = new int[0][0];
   // Check if input set has a size of 1 (trivial).
   if( n == 1 ) {
      L = new int[ (int) Math.pow( 2, n ) ][ n ]; // Size is [2^1][1].
      L[0][0] = 0;
      L[1][0] = 1;
   }
   else {
      // Generate recursively all subsets of a smaller input set (decrease-by-1).
      int[][] L1 = BinaryReflectedGrayCodeRec( n-1 );
      // Reverse list of newly generated subsets (for correct order of subsets).
      int[][] L1Rev = ReverseList( L1 );
      // Extend newly generated subsets including/excluding 1st set item.
      int[][] LPart1 = ExtendListAtFront( L1, 0 );
      int[][] LPart2 = ExtendListAtFront( L1Rev, 1 );
      // Merge extended lists.
      L = MergeLists( LPart1, LPart2 );
   }
   return L;
}
```

EXHAUSTIVE SEARCH

Exhaustive Search: To test each possible "value" sequentially in order to determine if it is the correct solution. The implicit assumption here is that we can check is a value is the correct solution, but we do not know how to "build" the solution step-by-step.

It is a special case of brute-force approach.

In using the exhaustive search approach, it is often necessary to find the problem solution (e.g., a special value) in a domain that grows exponentially or faster with the input size.

In many of these situations we have to deal with combinatorial objects:

- Permutations.
- Combinations.
- Subsets.

TRAVELING SALESMAN: PROBLEM

Problem: Given a set of n cities, the Traveling Salesman problem asks to find the shortest tour that visits each city exactly once before returning to the start city.

This problem can be conveniently modeled by a weighted graph, with the graph vertices representing the cities, and the edge weights specifying the distances.

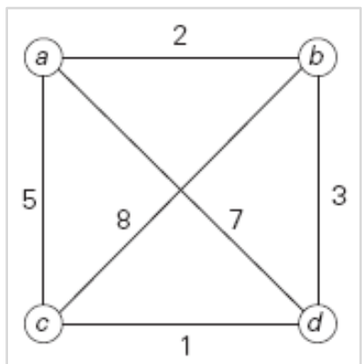


Figure 5. Weighted undirected complete graph representing the cities for the Traveling Salesman.

TRAVELING SALESMAN: ALGORITHM

Hamiltonian Circuit: A cycle that passes through all the vertices of the graph exactly once. That is, a sequence of n+1 adjacent vertices (v_{i0}, v_{i1}, … , $v_{i(n-1)}$, v_{i0}), where first and last vertices are the same, and all other vertices are distinct.

So, to solve the Traveling Salesman problem we have to find the shortest Hamiltonian circuit of the graph of cities. Further, we can assume, with no loss of generality, that all Hamiltonian circuits start and end at one particular vertex.

Traveling Salesman by Exhaustive Search

Step 1: Set city 0 to be the arbitrary start/end vertex.

Step 2: Generate all permutations of n-1 intermediate cities.

Step 3: Compute the tour-length of each permutation (including start/end vertex).

Step 4: Select the shortest tour/permutation.

TRAVELING SALESMAN: ALGORITHM 2

Example: Given a set of 4 cities { a, b, c, d } and a weighted (undirected complete) graph representing the cities/connections/distances, we can generate all possible tours (permutations) with vertex (a) as the start/end vertex, and then select the tour/permutation with the min weight/distance.

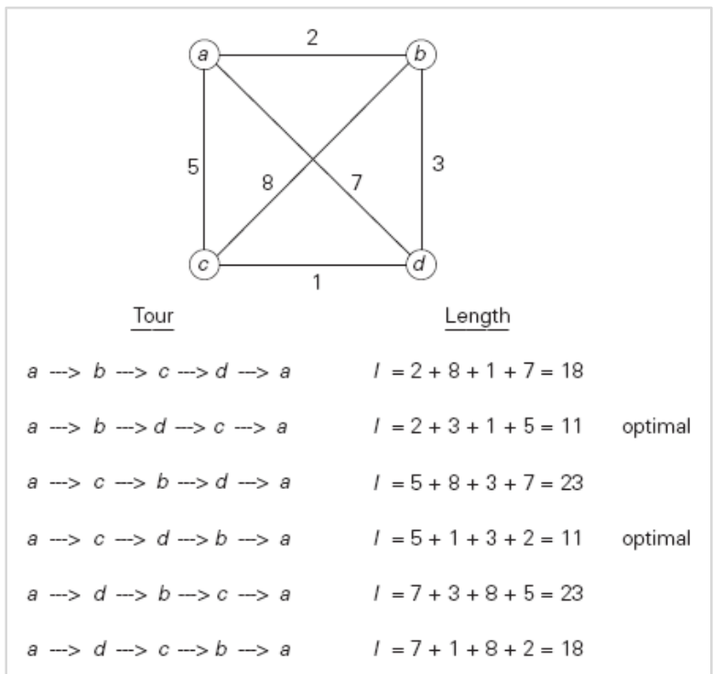


Figure 6. Solution of the Traveling Salesman problem, after modeling the cities with a weighted complete undirected graph, and finding the shortest tour.

TRAVELING SALESMAN: ANALYSIS

We can analyze the algorithm to solve the Traveling Salesman problem by exhaustive search, even without an actual implementation.

The input size is the number of cities in input (n).
The basic operation of the algorithm is computing a tour/permutation length.
The basic operation count C(n) depends only on the input size, because we have to process each permutation generated in any case.

$$C(n) = \text{permutations generated } \times 1 = P(n-1, n-1) = n-1!/1! \in \Theta(n!)$$

Note: We have to generate the permutation only of the n-1 cities (because the start/end city can be chosen arbitrarily.

Note: Please consider that in this case we considered all cities directly connected. In a real-world situation, the graph representing cities/connections/distances could be directed and non-complete: making the permutation generation more complex.

TRAVELING SALESMAN: OPTIMIZATION

Example: Considering the previous example (see Figure 6), a careful inspection of the permutations generated reveals half of tours that differ only by their direction.

So, we can cut the number of permutations by half, defining the tour/permutation direction (e.g., constraining the tour to move only in lexicographic order, allowing from (b) to (c) but forbidding from (c) to (b)).

If we do so, the total number of permutations needed is:

$$\mathrm{C}_{\mathbf{opt}}(n) = \frac{\mathbf{permutations\ num.}}{2} \times 1 = \frac{\mathrm{P}(n-1, n-1)}{2} = \frac{n-1!}{2} \in \Theta(n!)$$

Note: This optimization is not enough to improve the efficiency class of the algorithm, even if it reduces significantly the work done.

KNAPSACK: PROBLEM

Problem: Given n items of known weights { w_1, w_2, ... , w_n }, and values { v_1, v_2, ... , v_n }, and a knapsack of capacity W, find the most valuable subset that fit into the knapsack.

Clearly here, we can apply exhaustive search to check all possible arrangements of items in the knapsack (the power set of the set of items).

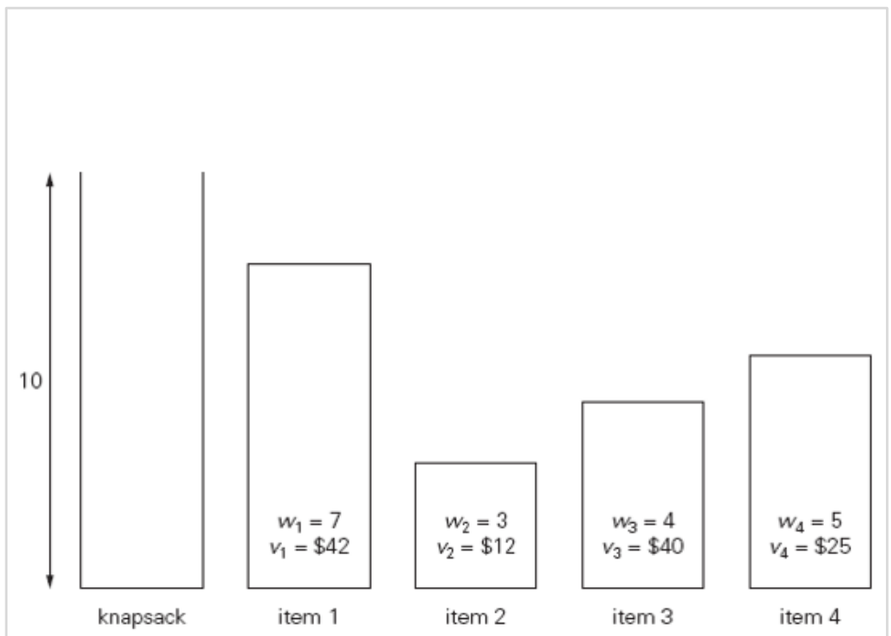


Figure 7. Example of the Knapsack problem with 4 items (n=4) and a knapsack capacity of 10 (W=10).

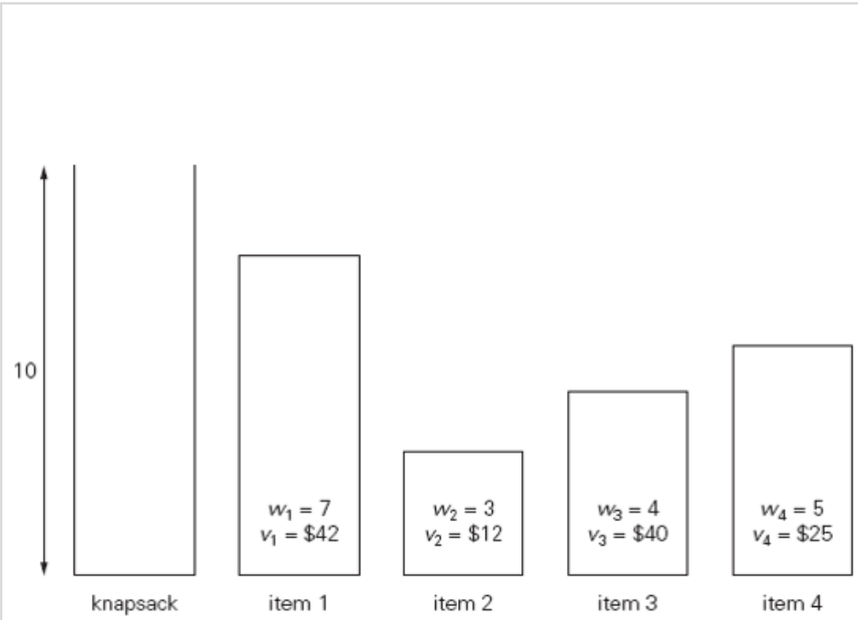
KNAPSACK: ALGORITHM

Knapsack by Exhaustive Search

Step 1: Generate all subsets of the set of n items.

Step 2: Discard all subset not-feasible (with subset weight > W).

Step 3: Among the feasible subsets, find the subset with greatest value.



Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Figure 8. An example of Knapsack problem (left), with its solution (bold) by exhaustive search (right).

KNAPSACK: ANALYSIS

We can analyze the algorithm to solve the Knapsack problem by exhaustive search, even without an actual implementation.

The input size is the number of items in input (n).
The basic operation of the algorithm is computing the value of a subset.
The basic operation count C(n) depends only on the input size, because we have to process each subset generated in any case.

$$C(n) = \text{subsets generated} \times 1 = 2^n \in \Theta(2^n)$$

Note: Both the Traveling Salesman and the Knapsack problems are examples of NP-hard problems: problems with no polynomial-time algorithm to solve them.

ASSIGNMENT: PROBLEM

Problem: Given n workers { w_1, w_2, ... , w_n } and n jobs { j_1, j_2, ... , j_n }, we need to assign each worker to execute a different job (1 worker per job, and 1 job per worker).
The cost of assigning the worker w_i to the job j_k is C[i,k].
The solution consists in finding the assignment with the minimum total cost.

A specific Assignment problem can be specified by its cost matrix C (see Table 1).

Table 1. An example of cost matrix C for an Assignment problem.

C[i,k]	J1	J2	J3
W1	7	1	4
W2	2	7	9
W3	2	5	4

ASSIGNMENT: PROBLEM 2

So, the problem is to select 1 element in each row of the cost matrix C so that all selected elements are in different columns and their total sum is the smallest possible.

We can describe feasible solutions to this problem as n-tuples $\langle x_1, \ldots, x_n \rangle$ where the i^{th} element (x_i), indicates the column (the job) of the element in the i^{th} row (the worker).
For example: $\langle 2, 1, 3 \rangle$ represents the solution (assignment) highlighted in Table 1.

Example: A simple Assignment problem is illustrated in Figure 9.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4>	cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3>	cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4>	cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2>	cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3>	cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2>	cost = 9 + 7 + 1 + 6 = 23

etc.

Figure 9. An example of Assignment problem solved by exhaustive search.

ASSIGNMENT: ALGORITHM

The Assignment problem requirements imply that there is a 1-to-1 correspondence between feasible assignments and permutations of the first n integers.

Assignment by Exhaustive Search

Step 1: Generate all permutations (assignments) of the integers { 1, 2, … , n }.

Step 2: Compute the total cost of each permutation (assignment).

Step 3: Select the permutation (assignment) with the minimum total cost.

ASSIGNMENT: ANALYSIS

We can analyze the algorithm to solve the Assignment problem by exhaustive search, even without an actual implementation.

The input size is the number of workers (or jobs) in input (n).
The basic operation of the algorithm is computing the total cost of an assignment.
The basic operation count C(n) depends only on the input size, because we have to process each permutation (assignment) generated in any case.

$$C(n) = \text{permutations (assignments) generated} \times 1 = n! \in \Theta(n!)$$

Note: Fortunately, we can solve the Assignment problem using a much more efficient algorithm called "the Hungarian method".

GRAPH TRAVERSALS

The exhaustive search approach is also applied to the main graph traversal algorithms that systematically process all vertices/nodes and edges/links of a graph:

- The depth-first search (DFS).
- The breadth-first search (BFS).

Note: Here we focus on the analysis of these two graph traversals algorithms, assuming that graphs (properties, terminology, and representations) have already been introduced.

DEPTH-FIRST SEARCH: ALGORITHM

Problem: Graph traversal (process all vertices/nodes and edges/links of a graph).

Depth-First Search The algorithm starts the graph traversal at an arbitrary vertex by marking it as visited. Then, on each iteration:

Step 1: The algorithm moves to an unvisited vertex (chosen arbitrarily) that is adjacent to the vertex where the algorithm is currently in, and marks the newly reached vertex as visited.

Step 2: Step 1 continues until the algorithm reaches a dead end: a vertex with no adjacent unvisited vertices.

Step 3: At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices (re-starting from Step 1). Otherwise, if backing up is impossible (dead end was start vertex), the algorithm halts.

DEPTH-FIRST SEARCH: IMPLEMENTATION

This is a Java implementation of Depth-First Search.

```
// Recursive DFS on a specific graph vertex.
private static int DFSRec( Graph g, int v, int c ) {
   c++; g.SetVisited( v, c ); // Mark input vertex with current traversal order.
   for( int w = 0; w < g.GetVNum(); w++ ) { // Iterate adjacents to input vertex.
      if( g.IsAdjacent( v, w ) ) { // If vertex not visited, start a DFS run.
         if( g.GetVisited( w ) == 0 ) { c = DFSRec( g, w, c ); }
      }
   }
   return c;
}

// Main DFS algorithm.
public static void DFS ( Graph g ) {
   int c = 0; // Init counter to keep track of the traversal order.
   for( int v = 0; v < g.GetVNum(); v++ ) { // Iterate among all graph vertices.
      // If vertex not visited, start a DFS at that vertex.
      if( g.GetVisited(v) == 0 ) { c = DFSRec( g, v, c ); }
   }
}
```

DEPTH-FIRST SEARCH: ANALYSIS

To analyze the DFS we have to consider the graph representation used (because that affects the performance of several algorithm operations).

This analysis shows that the DFS is quite efficient because it only requires a processing time proportional to the size of the graph representation used (data structure). Thus:

- If the graph representation is an adjacency matrix, the DFS time efficiency is in $\Theta(|V|^2)$ ($|V|$ is the number of vertices of the input graph, and $|V|^2$ is the number of cells of the adjacency matrix).
- If the graph representation is an adjacency list, the DFS time efficiency is in $\Theta(|V|+|E|)$ ($|V|$ is the number of vertices and $|E|$ is the number of edges of the input graph).

BREADTH-FIRST SEARCH: ALGORITHM

Problem: Graph traversal (process all vertices/nodes and edges/links of a graph).

Breadth-First Search The algorithm starts the graph traversal at an arbitrary vertex by marking it as visited. Then, on each iteration:

Step 1: The algorithm visits all the unvisited vertices (at once in arbitrary order) that are adjacent to the vertex where the algorithm is currently in ("expansion"), and marks all the newly reached vertices as visited.

Step 2: Then the algorithm moves on one of the visited vertices (not "expanded" yet), and perform again Step 1.

Step 3: The algorithm halts when all vertices have been "expanded".

BREADTH-FIRST SEARCH: IMPLEMENTATION

This is a Java implementation of Breadth-First Search.

```
public static void BFS( Graph g ) {
   int c = 0; // Init counter to track visit order during traversal.
   for( int v = 0; v < g.getVNum(); v++ ) { // Start BFS at vertex 0.
      if( g.getVisited(v) == 0 ) { // Check if current vertex is unvisited.
         c++; g.setVisited( v, c ); // Update counter and mark vertex as visited.
         Queue<Integer> queue = new LinkedList<Integer>(); queue.add(v);
         int front = -1; // Reset queue front to start new BFS.
         while( !queue.isEmpty() ) { // Perform BFS step until queue is empty.
            front = queue.peek();// Get current BFS vertex from queue front.
            for( int w = 0; w < g.getVNum(); w++ ) { // Visit adjacents/unvisiteds.
               if( ( g.isAdjacent( front, w ) ) && ( g.getVisited(w) == 0 ) ) {
                  c++; g.setVisited( w, c ); queue.add(w); // Mark as visited.
               }
            }
            queue.poll(); // Remove current vertex from queue (processing done).
         }
      }
   }
}
```

BREADTH-FIRST SEARCH: ANALYSIS

To analyze the BFS we have to consider the graph representation used (because that affects the performance of several algorithm operations).

This analysis shows that the BFS is quite efficient because it only requires a processing time proportional to the size of the graph representation used (data structure). Thus:

- If the graph representation is an adjacency matrix, the BFS time efficiency is in $\Theta(|V|^2)$ ($|V|$ is the number of vertices of the input graph, and $|V|^2$ is the number of cells of the adjacency matrix).
- If the graph representation is an adjacency list, the BFS time efficiency is in $\Theta(|V|+|E|)$ ($|V|$ is the number of vertices and $|E|$ is the number of edges of the input graph).

COMPARISON OF GRAPH TRAVERSALS

Table 2 shows a comparison of DFS and BFS graph traversals.

Table 2. Comparison of DFS and BFS graph traversals.

	DFS	BFS
Internal Data Structure	stack	queue
Main Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, min-edge path
Time Efficiency with Adjacency Matrix	$\Theta(\lvert V \rvert^2)$	$\Theta(\lvert V \rvert^2)$
Time Efficiency with Adjacency List	$\Theta(\lvert V \rvert + \lvert E \rvert)$	$\Theta(\lvert V \rvert + \lvert E \rvert)$