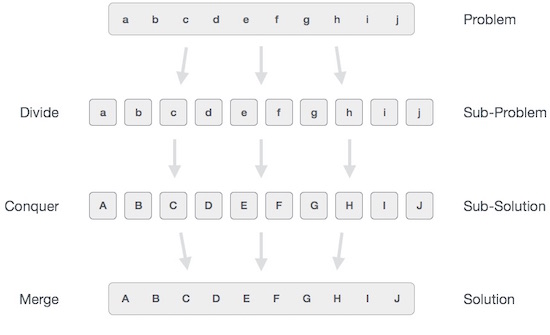
DIVIDE AND CONQUER

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub-problems into even smaller sub-problems, we may eventually reach at a stage where no more dividation is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of original problem.



Broadly, we can understand **divide-and-conquer** approach as three step process.

Divide/Break

* This step involves breaking the problem into smaller sub-problems. Sub-problems should represent as a part of original problem. This step generally takes recursive approach to divide the problem until no sub-problem is further dividable. At this stage, sub-problems become atomic in nature but still represents some part of actual problem.

Conquer/Solve

* This step receives lot of smaller sub-problem to be solved. Generally at this level, problems are considered 'solved' on their own.

Merge/Combine

* When the smaller sub-problems are solved, this stage recursively combines them until they formulate solution of the original problem.

This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Examples

The following computer algorithms are based on **divide-and-conquer** programming approach −

* Merge Sort
* Quick Sort
* Binary Search
* Strassen's Matrix Multiplication
* Closest pair (points)

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

GENERAL METHOD

Algorithm is a step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From data structure point of view, following are some important categories of algorithms −

* **Search** − Algorithm to search an item in a datastructure.
* **Sort** − Algorithm to sort items in certain order
* **Insert** − Algorithm to insert item in a datastructure
* **Update** − Algorithm to update an existing item in a data structure
* **Delete** − Algorithm to delete an existing item from a data structure

## Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the below mentioned characteristics −

* **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their input/outputs should be clear and must lead to only one meaning.
* **Input** − An algorithm should have 0 or more well defined inputs.
* **Output** − An algorithm should have 1 or more well defined outputs, and should match the desired output.
* **Finiteness** − Algorithms must terminate after a finite number of steps.
* **Feasibility** − Should be feasible with the available resources.
* **Independent** − An algorithm should have step-by-step directions which should be independent of any programming code.

## How to write an algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else) etc. These common constructs*can* be used to write an algorithm.

We write algorithms in step by step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

### Example

Let's try to learn algorithm-writing by using an example.

**Problem** − Design an algorithm to add two numbers and display result.

**step 1** − START

**step 2** − declare three integers **a**, **b** & **c**

**step 3** − define values of **a** & **b**

**step 4** − add values of **a** & **b**

**step 5** − store output of step 4 to **c**

**step 6** − print **c**

**step 7** − STOP

Algorithms tell the programmers how to code the program. Alternatively the algorithm can be written as −

**step 1** − START ADD

**step 2** − get values of **a** & **b**

**step 3** − c ← a + b

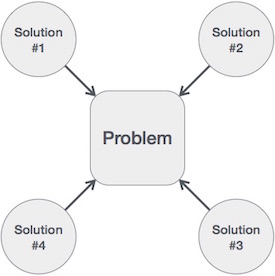
**step 4** − display c

**step 5** − STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy of the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. Next step is to analyze those proposed solution algorithms and implement the best suitable.

## Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation, as mentioned below −

* ***A priori* analysis** − This is theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation.
* ***A posterior* analysis** − This is empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn here **a priori** algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. Running time of an operation can be defined as no. of computer instructions executed per operation.

## Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

* **Time Factor** − The time is measured by counting the number of key operations such as comparisons in sorting algorithm
* **Space Factor** − The space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

## Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Space required by an algorithm is equal to the sum of the following two components −

* A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example simple variables & constant used, program size etc.
* A variable part is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stack space etc.

Space complexity **S(P)** of any algorithm **P** is **S(P) = C + SP(I)** Where **C** is the fixed part and **S(I)** is the variable part of the algorithm which depends on instance characteristic **I**. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

Here we have three variables A, B and C and one constant. Hence **S(P) = 1+3**. Now space depends on data types of given variables and constant types and it will be multiplied accordingly.

## Time Complexity

Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function **T(n)**, where **T(n)** can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is **T(n) = c\*n**, where **c** is the time taken for addition of two bits. Here, we observe that **T(n)** grows linearly as input size increases.

GREEDY METHOD

An algorithm is designed to achieve optimum solution for given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide optimum solution is chosen.

Greedy algorithms tries to find localized optimum solution which may eventually land in globally optimized solutions. But generally greedy algorithms do not provide globally optimized solutions.

Counting Coins

This problem is to count to a desired value by chosing least possible coins and greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of € 1, 2, 5 and 10 and we are asked to count € 18 then the greedy procedure will be −

* **1 −** Select one € 10 coin, remaining count is 8
* **2 −** Then select one € 5 coin, remaining count is 3
* **3 −** Then select one € 2 coin, remaining count is 1
* **3 −** And finally selection of one € 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins then necessary. For example − greedy approach will use 10 + 1 + 1 + 1 + 1 + 1 total 6 coins. Where the same problem could be solved by using only 3 coins (7 + 7 + 1)

Hence, we may conclude that greedy approach picks immediate optimized solution and may fail where global optimization is major concern.

**Algorithm:**

Step 1: Choose an input from the input set, based on

some criterion. If no more input exit.

Step 2: Check whether the chosen input yields to a

feasible solution. If no, discard the input and

goto step 1.

Step 3: Include the input into the solution vector and

update the objective function. Goto step 1.

Examples

Most networking algorithms uses greedy approach. Here is the list of few of them −

* Travelling Salesman Problem
* Prim's Minimal Spanning Tree Algorithm
* Kruskal's Minimal Spanning Tree Algorithm
* Dijkstra's Minimal Spanning Tree Algorithm
* Graph - Map Coloring
* Graph - Vertex Cover
* Knapsack Problem
* Job Scheduling Problem

These and there are lots of similar problems which uses greedy approach to find an optimum solution.

DYNAMIC PROGRAMMING

Dynamic programming approach is similar to divide and conquer in breaking down the problem in smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems which can be divided in similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that −

* The problem should be able to be divided in to smaller overlapping sub-problem.
* The optimum solution can be achieved by using optimum solution of smaller sub-problems.
* Dynamic algorithms use memoization.

Comparison

In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for overall optimization of the problem.

In contrast to divide and conquer algorithms, where solutions are combined to achieve overall solution, dynamic algorithms uses the output of smaller sub-problem and then try to optimize bigger sub-problem. Dynamic algorithms uses memoization to remember the output of already solved sub-problems.

Example

The following computer problems can be solved using dynamic programming approach −

* Fibonacci number series
* Knapsack problem
* Tower of Hanoi
* All pair shortest path by Floyd-Warshall
* Shortest path by Dijkstra
* Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And ofcourse, most of the times, referring to previous solution output is cheaper than re-computing in terms of CPU cycles.

Code optimization

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

* The output code must not, in any way, change the meaning of the program.
* Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
* Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

* At the beginning, users can change/rearrange the code or use better algorithms to write the code.
* After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
* While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

## Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

do

{

item = 10;

value = value + item;

} while(value<100);

This code involves repeated assignment of the identifier item, which if we put this way:

Item = 10;

do

{

value = value + item;

} while(value<100);

should not only save the CPU cycles, but can be used on any processor.

## Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

## Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

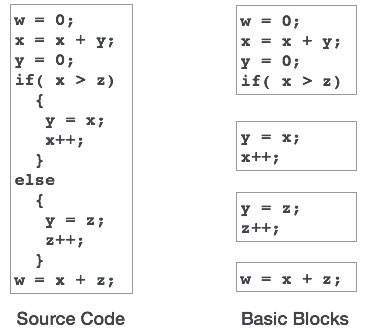
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

### Basic block identification

We may use the following algorithm to find the basic blocks in a program:

* Search header statements of all the basic blocks from where a basic block starts:
  + First statement of a program.
  + Statements that are target of any branch (conditional/unconditional).
  + Statements that follow any branch statement.
* Header statements and the statements following them form a basic block.
* A basic block does not include any header statement of any other basic block.

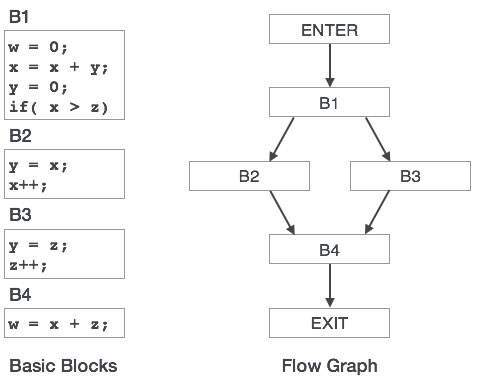
Basic blocks are important concepts from both code generation and optimization point of view.



Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

### Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.



## Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

* **Invariant code** : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
* **Induction analysis** : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
* **Strength reduction** : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x \* 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

## Dead-code Elimination

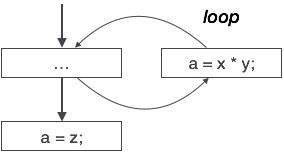
Dead code is one or more than one code statements, which are:

* Either never executed or unreachable,
* Or if executed, their output is never used.

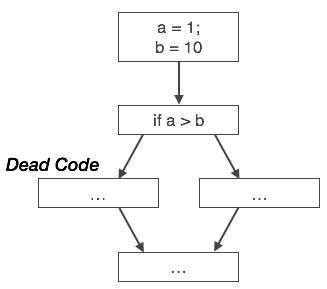
Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

### Partially dead code

There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.



The above control flow graph depicts a chunk of program where variable ‘a’ is used to assign the output of expression ‘x \* y’. Let us assume that the value assigned to ‘a’ is never used inside the loop.Immediately after the control leaves the loop, ‘a’ is assigned the value of variable ‘z’, which would be used later in the program. We conclude here that the assignment code of ‘a’ is never used anywhere, therefore it is eligible to be eliminated.



Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.

## Partial Redundancy

Redundant expressions are computed more than once in parallel path, without any change in operands.whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,

|  |  |
| --- | --- |
| Redundant Expression  [redundant expression] | Partially Redundant Expression  [partially redundant expression] |

Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique.

Another example of a partially redundant code can be:

If (condition)

{

a = y OP z;

}

else

{

...

}

c = y OP z;

We assume that the values of operands (**y** and **z**) are not changed from assignment of variable **a** to variable **c**. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

If (condition)

{

...

tmp = y OP z;

a = tmp;

...

}

else

{

...

tmp = y OP z;

}

c = tmp;

Here, whether the condition is true or false; y OP z should be computed only once.

### 

### And-Or Graphs

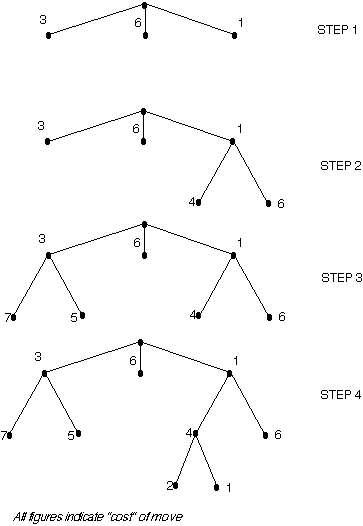
Useful for certain problems where

* The solution involves decomposing the problem into smaller problems.
* We then solve these smaller problems.

Here the alternatives often involve branches where some or all must be satisfied before we can progress.

For example if I want to learn to play a Frank Zappa guitar solo I could (Fig. [2.2.1](http://users.cs.cf.ac.uk/Dave.Marshall/AI2/node25.html#figandor_tree))

* Transcribe it from the CD. **OR**
* Buy the ``Frank Zappa Guitar Book'' **AND** Read it from there.



**Note** the use of arcs to indicate that one or more nodes must all be satisfied before the parent node is achieved. To find solutions using an And-Or GRAPH the best first algorithm is used as a basis with a modification to handle the set of nodes linked by the AND factor.