2. Pull out the sqrt call

The first optimization was to pull the sqrt call out of the limit test, just in case the compiler wasn't optimizing that correctly, this one is faster:

```
int is_prime(int n) {
  long lim = (int) sqrt(n);
  for (int i=2; i<=lim; i++) if (n%i == 0) return 0; return 1;}
```

3. Restatement of sqrt.

```
int is_prime(int n) {
  for (int i=2; i*i<=n; i++) if (n%i == 0) return 0;
  return 1;
}
```

4. We don't need to check even numbers

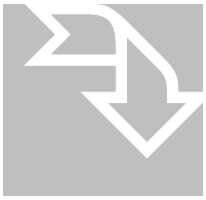
```
int is_prime(int n) {
  if (n == 1) return 0;          // 1 is NOT a prime
  if (n == 2) return 1;          // 2 is a prime
  if (n%2 == 0) return 0;        // NO prime is EVEN, except 2
  for (int i=3; i*i<=n; i+=2)    // start from 3, jump 2 numbers
    if (n%i == 0)                //  no need to check even numbers
      return 0;
  return 1;
}
```

5. Other prime properties

A (very) little bit of thought should tell you that no prime can end in 0,2,4,5,6, or 8, leaving only 1,3,7, and 9. It's fast & easy. Memorize this technique. It'll be very helpful for your programming assignments dealing with relatively small prime numbers (16-bit integer 1-32767). This divisibility check (step 1 to 5) will not be suitable for bigger numbers. First prime and the only even prime: 2.Largest prime in 32-bit integer range: 2^31 - 1 = 2,147,483,647

6. Divisibility check using smaller primes below sqrt(N):

Actually, we can improve divisibility check for bigger numbers. Further investigation concludes that a number N is a prime if and only if no primes below sqrt(N) can divide N.



How to do this ?

1. Create a large array. How large?
2. Suppose max primes generated will not greater than 2^31-1 (2,147,483,647), maximum 32-bit integer.
3. Since you need smaller primes below sqrt(N), you only need to store primes from 1 to sqrt(2^31)
4. Quick calculation will show that of sqrt(2^31) = 46340.95.
5. After some calculation, you'll find out that there will be at most 4792 primes in the range 1 to 46340.95. So you only need about array of size (roughly) 4800 elements.
6. Generate that prime numbers from 1 to 46340.955. This will take time, but when you already have those 4792 primes in hand, you'll be able to use those values to determine whether a bigger number is a prime or not.
7. Now you have 4792 first primes in hand. All you have to do next is to check whether a big number N a prime or not by dividing it with small primes up to sqrt(N). If you can find at least one small primes can divide N, then N is not prime, otherwise N is prime.

Fermat Little Test:

This is a probabilistic algorithm so you cannot guarantee the possibility of getting correct answer. In a range as big as 1-1000000, Fermat Little Test can be fooled by (only) 255 Carmichael numbers (numbers that can fool Fermat Little Test, see Carmichael numbers above). However, you can do multiple random checks to increase this probability.

Fermat Algorithm

```
If 2^N modulo N = 2 then N has a high probability to be a prime number.
```

Example:
let N=3 (we know that 3 is a prime).
2^3 mod 3 = 8 mod 3 = 2, then N has a high probability to be a prime number... and in fact, it is really prime.

Another example:
let N=11 (we know that 11 is a prime).
2^11 mod 11 = 2048 mod 11 = 2, then N has a high probability to be a prime number... again, this is also really prime.

Sieve of Eratosthenes:

Sieve is the best prime generator algorithm. It will generate a list of primes very quickly, but it will need a very big memory. You can use Boolean flags to do this (Boolean is only 1 byte).

Algorithm for Sieve of Eratosthenes to find the prime numbers within a range L,U (inclusive), where must be L<=U.

```
void sieve(int L,int U) {
  int i,j,d;
  d=U-L+1; /* from range L to U, we have d=U-L+1 numbers. */
  /* use flag[i] to mark whether (L+i) is a prime number or not. */

  bool *flag=new bool[d];
  for (i=0;i<d;i++) flag[i]=true; /* default: mark all to be true */

  for (i=(L%2!=0);i<d;i+=2) flag[i]=false;


  /* sieve by prime factors staring from 3 till sqrt(U) */
  for (i=3;i<=sqrt(U);i+=2) {
    if (i>L && !flag[i-L]) continue;

    /* choose the first number to be sieved -- >=L,
       divisible by i, and not i itself! */
    j=L/i*i;    if (j<L) j+=i;
    if (j==i) j+=i; /* if j is a prime number, have to start form next
one */

    j-=L; /* change j to the index representing j */
    for (;j<d;j+=i) flag[j]=false;
  }

  if (L<=1) flag[1-L]=false;
  if (L<=2) flag[2-L]=true;

  for (i=0;i<d;i++) if (flag[i]) cout << (L+i) << " ";
  cout << endl;
}
```

CHAPTER 8 SORTING

Definition of a sorting problem

Input: A sequence of N numbers (a_1,a_2,...,a_N)
Output: A permutation ($a_{1'}$,$a_{2'}$,...,$a_{N'}$) of the input sequence such that $a_{1'} <= a_{2'} <= ... <= a_{N'}$

Things to be considered in Sorting

These are the difficulties in sorting that can also happen in real life:

A. Size of the list to be ordered is the main concern. Sometimes, the computer emory is not sufficient to store all data. You may only be able to hold part of the data inside the computer at any time, the rest will probably have to stay on disc or tape. This is known as the problem of external sorting. However, rest assured, that almost all programming contests problem size will never be extremely big such that you need to access disc or tape to perform external sorting... (such hardware access is usually forbidden during contests).
B. Another problem is the *stability* of the sorting method. Example: suppose you are an airline. You have a list of the passengers for the day's flights. Associated to each passenger is the number of his/her flight. You will probably want to sort the list into alphabetical order. No problem... Then, you want to re-sort the list by flight number so as to get lists of passengers for each flight. Again, "no problem"... - except that it would be very nice if, for each flight list, the names were still in alphabetical order. This is the problem of stable sorting.
C. To be a bit more mathematical about it, suppose we have a list of items $\{x_i\}$ with x_a equal to x_b as far as the sorting comparison is concerned and with x_a before x_b in the list. The sorting method is stable if x_a is sure to come before x_b in the sorted list.

Finally, we have the problem of *key sorting*. The individual items to be sorted might be very large objects (e.g. complicated record cards). All sorting methods naturally involve a lot of moving around of the things being sorted. If the things are very large this might take up a lot of computing time -- much more than that taken just to switch two integers in an array.

Comparison-based sorting algorithms

Comparison-based sorting algorithms involves comparison between two object a and b to determine one of the three possible relationship between them: less than, equal, or greater

than. These sorting algorithms are dealing with how to use this comparison effectively, so that we minimize the amount of such comparison. Lets start from the most naive version to the most sophisticated comparison-based sorting algorithms.

Bubble Sort

Speed: O(n^2), extremely slow
Space: The size of initial array
Coding Complexity: Simple

This is the simplest and (unfortunately) the worst sorting algorithm. This sort will do double pass on the array and swap 2 values when necessary.

```
BubbleSort(A)
  for i <- length[A]-1 down to 1
    for j <- 0 to i-1
      if (A[j] > A[j+1]) // change ">" to "<" to do a descending sort
        temp <- A[j]
        A[j] <- A[j+1]
        A[j+1] <- temp
```

Slow motion run of Bubble Sort (**Bold** == sorted region):

5 2 3 1 4
2 3 1 4 **5**
2 1 3 **4 5**
1 2 **3 4 5**
1 **2 3 4 5**
1 2 3 4 5 >> done

TEST YOUR BUBBLE SORT KNOWLEDGE

Solve UVa problems related with Bubble sort:

299 - Train Swapping
612 - DNA Sorting
10327 - Flip Sort

Quick Sort

Speed: O(n log n), one of the best sorting algorithm.
Space: The size of initial array
Coding Complexity: Complex, using Divide & Conquer approach

One of the best sorting algorithm known. Quick sort use Divide & Conquer approach and partition each subset. Partitioning a set is to divide a set into a collection of mutually disjoint sets. This sort is much quicker compared to "stupid but simple" bubble sort. Quick sort was invented by C.A.R Hoare.

Quick Sort - basic idea

Partition the array in O(n)
Recursively sort left array in O(log2 n) best/average case
Recursively sort right array in O(log2 n) best/average case

Quick sort pseudo code:

```
QuickSort(A,p,r)
  if p < r
    q <- Partition(A,p,r)
    QuickSort(A,p,q)
    QuickSort(A,q+1,r)
```

Quick Sort for C/C++ User

C/C++ standard library <stdlib.h> contains qsort function.

This is not the best quick sort implementation in the world but it fast enough and VERY EASY to be used... therefore if you are using C/C++ and need to sort something, you can simply call this built in function:

```
qsort(<arrayname>,<size>,sizeof(<elementsize>),compare_function);
```

The only thing that you need to implement is the compare_function, which takes in two arguments of type "const void", which can be cast to appropriate data structure, and then return one of these three values:

- negative, if a should be before b
- 0, if a equal to b
- positive, if a should be after b

1. Comparing a list of integers

simply cast a and b to integers
if x < y, x-y is negative, x == y, x-y = 0, x > y, x-y is positive
x-y is a shortcut way to do it :)
reverse *x - *y to *y - *x for sorting in decreasing order

```
int compare_function(const void *a,const void *b) {
  int *x = (int *) a;
  int *y = (int *) b;
  return *x - *y;
}
```

2. Comparing a list of strings

For comparing string, you need strcmp function inside string.h lib. strcmp will by default return -ve,0,ve appropriately... to sort in reverse order, just reverse the sign returned by strcmp

```
#include <string.h>

int compare_function(const void *a,const void *b) {
   return (strcmp((char *)a,(char *)b));
}
```

3. Comparing floating point numbers

```
int compare_function(const void *a,const void *b) {
  double *x = (double *) a;
  double *y = (double *) b;
  // return *x - *y; // this is WRONG...
  if (*x < *y) return -1;
  else if (*x > *y) return 1;  return 0;
}
```

4. Comparing records based on a key

Sometimes you need to sort a more complex stuffs, such as record. Here is the simplest way to do it using qsort library

```
typedef struct {
  int key;
  double value;
} the_record;
```

```
int compare_function(const void *a,const void *b) {
  the_record *x = (the_record *) a;
  the_record *y = (the_record *) b;
  return x->key - y->key;
}
```

Multi field sorting, advanced sorting technique

Sometimes sorting is not based on one key only.

For example sorting birthday list. First you sort by month, then if the month ties, sort by date (obviously), then finally by year.

For example I have an unsorted birthday list like this:

```
24 - 05 - 1982 - Sunny
24 - 05 - 1980 - Cecilia
31 - 12 - 1999 - End of 20th century
01 - 01 - 0001 - Start of modern calendar
```

I will have a sorted list like this:

```
01 - 01 - 0001 - Start of modern calendar
24 - 05 - 1980 - Cecilia
24 - 05 - 1982 - Sunny
31 - 12 - 1999 - End of 20th century
```

To do multi field sorting like this, traditionally one will choose multiple sort using sorting algorithm which has "stable-sort" property.

The better way to do multi field sorting is to modify the compare_function in such a way that you break ties accordingly... I'll give you an example using birthday list again.

```
typedef struct {
  int day,month,year;
  char *name;
} birthday;

int compare_function(const void *a,const void *b) {
  birthday *x = (birthday *) a;
  birthday *y = (birthday *) b;

  if (x->month != y->month) // months different
    return x->month - y->month; // sort by month
```

```
  else { // months equal..., try the 2nd field... day
    if (x->day != y->day) // days different

      return x->day - y->day; // sort by day
    else // days equal, try the 3rd field... year
      return x->year - y->year; // sort by year
  }
}
```

TEST YOUR MULTI FIELD SORTING KNOWLEDGE

10194 - Football (aka Soccer)

Linear-time Sorting

a. Lower bound of comparison-based sort is O(n log n)

The sorting algorithms that we see above are comparison-based sort, they use comparison function such as <, <=, =, >, >=, etc to compare 2 elements. We can model this comparison sort using decision tree model, and we can proof that the shortest height of this tree is **O(n log n)**.

b. Counting Sort

For Counting Sort, we assume that the numbers are in the range [0..k], where k is at most O(n). We set up a counter array which counts how many duplicates inside the input, and the reorder the output accordingly, without any comparison at all. Complexity is ***O(n+k)***.

c. Radix Sort

For Radix Sort, we assume that the input are ***n d***-digits number, where ***d*** is reasonably limited.

Radix Sort will then sorts these number digit by digit, starting with the least significant digit to the most significant digit. It usually use a stable sort algorithm to sort the digits, such as Counting Sort above.

Example:

input:
321
257

113
622

sort by third (last) digit:
32**1**
62**2**
11**3**
25**7**

after this phase, the third (last) digit is sorted.

sort by second digit:
1**1**3
3**2**1
6**2**2
2**5**7
after this phase, the second and third (last) digit are sorted.

sort by second digit:
113
257
321
622
after this phase, all digits are sorted.

For a set of ***n*** ***d***-digits numbers, we will do ***d*** pass of counting sort which have complexity ***O(n+k)***, therefore, the complexity of Radix Sort is ***O(d(n+k))***.

CHAPTER 9 SEARCHING

Searching is very important in computer science. It is very closely related to sorting. We usually search after sorting the data. Remember Binary Search? This is the best example of an algorithm which utilizes both Sorting and Searching.[2]

Search algorithm depends on the data structure used. If we want to search a graph, then we have a set of well known algorithms such as DFS, BFS, etc

Binary Search

The most common application of binary search is to find a specific value in a sorted list. The search begins by examining the value in the center of the list; because the values are sorted, it then knows whether the value occurs before or after the center value, and searches through the correct half in the same way. Here is simple pseudocode which determines the index of a given value in a sorted list a between indices left and right:

```
function binarySearch(a, value, left, right)
    if right < left
        return not found
    mid := floor((left+right)/2)
    if a[mid] = value
        return mid
    if value < a[mid]
        binarySearch(a, value, left, mid-1)
    else
        binarySearch(a, value, mid+1, right)
```

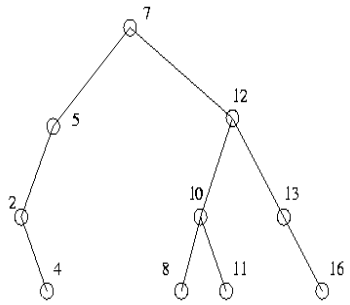
In both cases, the algorithm terminates because on each recursive call or iteration, the range of indexes right minus left always gets smaller, and so must eventually become negative.

Binary search is a logarithmic algorithm and executes in O(log n) time. Specifically, 1 + log2N iterations are needed to return an answer. It is considerably faster than a linear search. It can be implemented using recursion or iteration, as shown above, although in many languages it is more elegantly expressed recursively.

Binary Search Tree

Binary Search Tree (BST) enable you to search a collection of objects (each with a real or integer value) quickly to determine if a given value exists in the collection.

Basically, a binary search tree is a node-weighted, rooted binary ordered tree. That collection of adjectives means that each node in the tree might have no child, one left child, one right child, or both left and right child. In addition, each node has an object associated with it, and the weight of the node is the value of the object.



The binary search tree also has the **property** that each node's left child and descendants of its left child have a value less than that of the node, and each node's right child and its descendants have a value greater or equal to it. Binary Search Tree

The nodes are generally represented as a structure with four fields, a pointer to the node's left child, a pointer to the node's right child, the weight of the object stored at this node, and a pointer to the object itself. Sometimes, for easier access, people add pointer to the parent too.

Why are Binary Search Tree useful?

Given a collection of n objects, a binary search tree takes only O(height) time to find an objects, assuming that the tree is not really poor (unbalanced), O(height) is O(log n). In addition, unlike just keeping a sorted array, inserting and deleting objects only takes O(log n) time as well. You also can get all the keys in a Binary Search Tree in a sorted order by traversing it using O(n) inorder traversal.

Variations on Binary Trees

There are several variants that ensure that the trees are never poor. Splay trees, Red-black trees, B-trees, and AVL trees are some of the more common examples. They are all much more complicated to code, and random trees are generally good, so it's generally not worth it.

Tips: If you're concerned that the tree you created might be bad (it's being created by inserting elements from an input file, for example), then randomly order the elements before insertion.

Dictionary

A dictionary, or hash table, stores data with a very quick way to do lookups. Let's say there is a collection of objects and a data structure must quickly answer the question: 'Is

this object in the data structure?' (e.g., is this word in the dictionary?). A hash table does this in less time than it takes to do binary search.

The idea is this: find a function that maps the elements of the collection to an integer between 1 and x (where x, in this explanation, is larger than the number of elements in your collection). Keep an array indexed from 1 to x, and store each element at the position that the function evaluates the element as. Then, to determine if something is in your collection, just plug it into the function and see whether or not that position is empty. If it is not check the element there to see if it is the same as the something you're holding,

For example, presume the function is defined over 3-character words, and is (first letter + (second letter * 3) + (third letter * 7)) mod 11 (A=1, B=2, etc.), and the words are 'CAT', 'CAR', and 'COB'. When using ASCII, this function takes 'CAT' and maps it to 3, maps 'CAR' to 0, and maps 'COB' to 7, so the hash table would look like this:

```
0: CAR
1
2
3: CAT
4
5
6
7: COB
8
9
10
```

Now, to see if 'BAT' is in there, plug it into the hash function to get 2. This position in the hash table is empty, so it is not in the collection. 'ACT', on the other hand, returns the value 7, so the program must check to see if that entry, 'COB', is the same as 'ACT' (no, so 'ACT' is not in the dictionary either). In the other hand, if the search input is 'CAR', 'CAT', 'COB', the dictionary will return true.

Collision Handling

This glossed over a slight problem that arises. What can be done if two entries map to the same value (e.g.,we wanted to add 'ACT' and 'COB')? This is called a collision. There are couple ways to correct collisions, but this document will focus on one method, called chaining.

Instead of having one entry at each position, maintain a linked list of entries with the same hash value. Thus, whenever an element is added, find its position and add it to the

beginning (or tail) of that list. Thus, to have both 'ACT' and 'COB' in the table, it would look something like this:

```
0: CAR
1
2
3: CAT
4
5
6
7: COB -> ACT
8
9
10
```

Now, to check an entry, all elements in the linked list must be examined to find out the element is not in the collection. This, of course, decreases the efficiency of using the hash table, but it's often quite handy.

Hash Table variations

It is often quite useful to store more information that just the value. One example is when searching a small subset of a large subset, and using the hash table to store locations visited, you may want the value for searching a location in the hash table with it.

CHAPTER 10 GREEDY ALGORITHMS

Greedy algorithms are algorithms which follow the problem solving meta-heuristic of making the locally optimum choice at each stage with the hope of finding the global optimum. For instance, applying the greedy strategy to the traveling salesman problem yields the following algorithm: "At each stage visit the nearest unvisited city to the current city".[Wiki Encyclopedia]

Greedy algorithms do not consistently find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. For example, all known greedy algorithms for the graph coloring problem and all other NP-complete problems do not consistently find optimum solutions. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.

If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice. Examples of such greedy algorithms are Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees and the algorithm for finding optimum Huffman trees. The theory of matroids, as well as the even more general theory of greedoids, provide whole classes of such algorithms.

In general, greedy algorithms have five pillars:

- A candidate set, from which a solution is created
- A selection function, which chooses the best candidate to be added to the solution
- A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
- An objective function, which assigns a value to a solution, or a partial solution, and
- A solution function, which will indicate when we have discovered a complete solution

Barn Repair

There is a long list of stalls, some of which need to be covered with boards. You can use up to N (1 <= N <= 50) boards, each of which may cover any number of consecutive stalls. Cover all the necessary stalls, while covering as few total stalls as possible. How will you solve it?

The basic idea behind greedy algorithms is to build large solutions up from smaller ones. Unlike other approaches, however, greedy algorithms keep only the best solution they find as they go along. Thus, for the sample problem, to build the answer for N = 5, they find the best solution for N = 4, and then alter it to get a solution for N = 5. No other solution for N = 4 is ever considered.

Greedy algorithms are fast, generally linear to quadratic and require little extra memory. Unfortunately, they usually aren't correct. But when they do work, they are often easy to implement and fast enough to execute.

Problems with Greedy algorithms

There are two basic problems to greedy algorithms.

1. How to Build

How does one create larger solutions from smaller ones? In general, this is a function of the problem. For the sample problem, the most obvious way to go from four boards to five boards is to pick a board and remove a section, thus creating two boards from one. You should choose to remove the largest section from any board which covers only stalls which don't need covering (so as to minimize the total number of stalls covered).

To remove a section of covered stalls, take the board which spans those stalls, and make into two boards: one of which covers the stalls before the section, one of which covers the stalls after the second.

2. Does it work?

The real challenge for the programmer lies in the fact that greedy solutions don't always work. Even if they seem to work for the sample input, random input, and all the cases you can think of, if there's a case where it won't work, at least one (if not more!) of the judges' test cases will be of that form.

For the sample problem, to see that the greedy algorithm described above works, consider the following:

Assume that the answer doesn't contain the large gap which the algorithm removed, but does contain a gap which is smaller. By combining the two boards at the end of the smaller gap and splitting the board across the larger gap, an answer is obtained which uses as many boards as the original solution but which covers fewer stalls. This new answer is better, so therefore the assumption is wrong and we should always choose to remove the largest gap.

If the answer doesn't contain this particular gap but does contain another gap which is just as large, doing the same transformation yields an answer which uses as many boards and covers as many stalls as the other answer. This new answer is just as good as the original solution but no better, so we may choose either.

Thus, there exists an optimal answer which contains the large gap, so at each step, there is always an optimal answer which is a superset of the current state. Thus, the final answer is optimal.

If a greedy solution exists, use it. They are easy to code, easy to debug, run quickly, and use little memory, basically defining a good algorithm in contest terms. The only missing element from that list is correctness. If the greedy algorithm finds the correct answer, go for it, but don't get suckered into thinking the greedy solution will work for all problems.

Sorting a three-valued sequence

You are given a three-valued (1, 2, or 3) sequence of length up to 1000. Find a minimum set of exchanges to put the sequence in sorted order.

The sequence has three parts: the part which will be 1 when in sorted order, 2 when in sorted order, and 3 when in sorted order. The greedy algorithm swaps as many as possible of the 1's in the 2 part with 2's in the 1 part, as many as possible 1's in the 3 part with 3's in the 1 part, and 2's in the 3 part with 3's in the 2 part. Once none of these types remains, the remaining elements out of place need to be rotated one way or the other in sets of 3. You can optimally sort these by swapping all the 1's into place and then all the 2's into place.

Analysis: Obviously, a swap can put at most two elements in place, so all the swaps of the first type are optimal. Also, it is clear that they use different types of elements, so there is no ``interference'' between those types. This means the order does not matter. Once those swaps have been performed, the best you can do is two swaps for every three elements not in the correct location, which is what the second part will achieve (for example, all the 1's are put in place but no others; then all that remains are 2's in the 3's place and vice-versa, and which can be swapped).

Topological Sort

Given a collection of objects, along with some ordering constraints, such as "A must be before B," find an order of the objects such that all the ordering constraints hold.

Algorithm: Create a directed graph over the objects, where there is an arc from A to B if "A must be before B." Make a pass through the objects in arbitrary order. Each time you find an object with in-degree of 0, greedily place it on the end of the current ordering, delete all of its out-arcs, and recurse on its (former) children, performing the same check. If this algorithm gets through all the objects without putting every object in the ordering, there is no ordering which satisfies the constraints.

TEST YOUR GREEDY KNOWLEDGE

Solve UVa problems related which utilizes Greedy algorithms:

10020 - Minimal Coverage
10340 - All in All
10440 - Ferry Loading (II)

CHAPTER 11 DYNAMIC PROGRAMMING

Dynamic Programming (shortened as DP) is a programming technique that can dramatically reduces the runtime of some algorithms (but not all problem has DP characteristics) from exponential to polynomial. Many (and still increasing) real world problems only solvable within reasonable time using DP.

To be able to use DP, the original problem must have:

1. **Optimal sub-structure** property:
   Optimal solution to the problem contains within it optimal solutions to sub-problems
2. **Overlapping sub-problems** property
   We accidentally recalculate the same problem twice or more.

There are 2 types of DP: We can either build up solutions of sub-problems from small to large (bottom up) or we can save results of solutions of sub-problems in a table (top down + memoization).

Let's start with a sample of Dynamic Programming (DP) technique. We will examine the simplest form of overlapping sub-problems. Remember Fibonacci? A popular problem which creates a lot of redundancy if you use standard recursion $f_n = f_{n-1} + f_{n-2}$.

Top-down Fibonacci DP solution will record the each Fibonacci calculation in a table so it won't have to re-compute the value again when you need it, a simple table-lookup is enough (memorization), whereas Bottom-up DP solution will build the solution from smaller numbers.

Now let's see the comparison between Non-DP solution versus DP solution (both bottom-up and top-down), given in the C source code below, along with the appropriate comments

```
#include <stdio.h>

#define MAX 20 // to test with bigger number, adjust this value

int memo[MAX]; // array to store the previous calculations

// the slowest, unnecessary computation is repeated
int Non_DP(int n) {
  if (n==1 || n==2)
    return 1;
```

```
else
    return Non_DP(n-1) + Non_DP(n-2);
}

// top down DP
int DP_Top_Down(int n) {
  // base case
  if (n == 1 || n == 2)
    return 1;

  // immediately return the previously computed result
  if (memo[n] != 0)
    return memo[n];

  // otherwise, do the same as Non_DP
  memo[n] = DP_Top_Down(n-1) + DP_Top_Down(n-2);
  return memo[n];
}

// fastest DP, bottom up, store the previous results in array
int DP_Bottom_Up(int n) {
  memo[1] = memo[2] = 1; // default values for DP algorithm

  // from 3 to n (we already know that fib(1) and fib(2) = 1
  for (int i=3; i<=n; i++)
    memo[i] = memo[i-1] + memo[i-2];

  return memo[n];
}

void main() {
  int z;

  // this will be the slowest
  for (z=1; z<MAX; z++) printf("%d-",Non_DP(z));
  printf("\n\n");

  // this will be much faster than the first
  for (z=0; z<MAX; z++) memo[z] = 0;
  for (z=1; z<MAX; z++) printf("%d-",DP_Top_Down(z));
  printf("\n\n");

  /* this normally will be the fastest */
  for (z=0; z<MAX; z++) memo[z] = 0;
  for (z=1; z<MAX; z++) printf("%d-",DP_Bottom_Up(z));
  printf("\n\n");
}
```