Part 4: Brute Force Algorithms

Contents

- Brute Force Algorithm Design p.2
- Selection Sort p.4
- String Matching p.8
- Closest-Pair Problem p.11
- Knapsack Problem p.14
- Job Assignment Problem p.16

[document finalized]

Brute Force Algorithm Design

"Brute force" is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

The "force" implied by this definition is that of a computer and not that of one's intellect. There is a tradeoff between the time the program runs to produce an answer and the time it takes us to write a correct program.

For instance, if we anticipate that we only need to run a program a few times and the simplest solution we can think of runs sufficiently fast, why not using it? We will make less errors implementing it, and the total time (designing + proving correctness + implementing + testing + running the program) may be minimal.

On the other hand, if we want to use a program frequently or want to apply it to large inputs we are interested in finding efficient solutions, i.e. we may want to invest more time in finding a faster algorithm, that for instance decreases the runtime from $\Theta(n^2)$ to $\Theta(n \log n)$ (sorting), or better yet, from $\Omega(n!)$ to $\Theta(n^3)$ (job assignment problem). Here we discuss several examples

of brute force algorithm design.

Selection Sort

Perhaps the simplest way to sort n numbers (or **keys** in general) in non-decreasing order is based on this idea: what property does the first number in a sorted sequence have? It is the smallest ... So we could first find the smallest key in the sequence and then swap it with the first key. Then what's left is a sorting problem which is smaller than the original one: we need to sort the remaining $n-1$ numbers, etc. This sorting procedure is called "selection sort", because in each step we select the minimum, swap, and continue.

Sample run:

```
i
3 0 4 2 1 6 5
  m

  i
0 3 4 2 1 6 5
      m

0 1 4 2 3 6 5
    i m

0 1 2 4 3 6 5
      i m

0 1 2 3 4 6 5
        i
        m

0 1 2 3 4 6 5
          i m

0 1 2 3 4 5 6   done
            i
```

Pseudo Code:

```
// sort n > 0 keys in non-decreasing order
function SelectionSort(A[0..n-1])
for i <- 0 to n-2 do
  // find minimum key in A[i..n-1]
  m <- i            // index of minimum key
  for j <- i+1 to n-1 do
    if A[j] < A[m] then  // (*)
      m <- j        // found smaller value
    end
  end

  // swap minimal key with current one
  t <- A[i]
  A[i] <- A[m]
  A[m] <- t
end
```

Correctness:

Above program terminates for all inputs because it only features for-loops, which we know always stop regardless of the bounds. Finding invariants for both loops based on the algorithm's description above is left as an exercise.

Runtime Analysis:

To simplify the runtime analysis we concentrate on how often the innermost loop body (*) is executed. Every other part is executed at most as often.

In the outer loop, i counts from 0 to $n-2$, that's less than n times, and the inner loop is executed also less than n times for each i.

Thus, the inner loop is executed less than n^2 times in total, i.e., worst-case runtime $T(n) \in O(n^2)$.

The exact number of times (*) is executed is the same as for the uniqueness algorithm we have seen before $\rightsquigarrow\ T(n) \in \Theta(n^2)$.

So, the runtime of selection sort is quadratic in n, which is SLOW:

n	n*n
10	100
100	10000
1000	1000000
10000	100000000

Imagine having to sort 1000000 numbers ... Even on

today's fast desktop computers this would take more than 1000 seconds when using selection sort.

Can we do better?

Yes — later we will study sorting algorithms with worst-case runtime $\Theta(n \log n)$.

It is often the case that sophisticated algorithms that asymptotically are more efficient than basic algorithms are only truly faster beyond a certain input size. This means that sometimes hybrid algorithms, that use basic approaches for small input sizes and sophisticated methods for larger inputs, are the overall winners.

String Matching

Searching for words in texts is a very common task. Applications range from text editors to matching genes.

The problem is this: given a text character string T and a pattern character string P, determine whether P occurs in T and if so, report the first matching location.

The straightforward "brute force" solution is to shift P from left to right and to see whether it matches T at the current location.

Example:

```
i= 012345678901234567
             11111111
T= NOBODY_NOTICED_HIM
P= NOT  no
    NOT  no
     NOT  no
      NOT  no
       NOT  no
        NOT  no
         NOT  no
          NOT  yes! return 7
```

```
// returns smallest index of pattern P
// in text T if it occurs, and -1 otherwise
function StringMatch(T[0..n-1], P[0..m-1])
for i <- 0 to n-m do
  // does P match T at location i?
  j <- 0
  while j < m and P[j] = T[i+j] do
    j <- j + 1
  end
  if j = m then
    return i     // yes, return location
  end
end
return -1        // pattern not found
```

In the worst case, when P does not occur, the runtime of `StringMatch` is $\Theta(nm)$, because the first loop runs to completion ($n - m + 1$ iterations), and the inner loop always iterates m times.

When m is small, this algorithm is very efficient. However, the runtime degenerates to $\Theta(n^2)$ in case m growths linearly in n. This may be unacceptable. Is there a faster matching algorithm?

Yes, several.

The Knuth-Morris-Pratt (KMP) algorithm, for instance, takes $\Theta(m)$ time for preprocessing P, followed by $\Theta(n)$ operations to search for P in T using the result of the preprocessing step. Total runtime: $\Theta(n + m)$, which depending on n, m can be much smaller than $\Theta(nm)$.

The closest pair problem

Example from computational geometry:

Given n points in the plane, determine the pair of points that is closest to each other.

How can we determine such pair? Following the "brute force" design approach, we could simply enumerate all pairs of points and keep track of the minimal encountered distance.

Recall: the distance d of two points (x_1, y_1) and (x_2, y_2) in the plane is given by $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Because the square root function is monotonically increasing (for $a, b \geq 0,\ a < b \Leftrightarrow \sqrt{a} < \sqrt{b}$), we actually only need to compare squared distances, thereby avoiding the costly and potentially inaccurate square root computation.

These ideas lead to the following pseudo code:

```
function ClosestPair(X[0..n-1], Y[0..n-1])
dsmin <- oo   // current smallest squared distance
for i <- 0 to n-2       // look at all point pairs
  for j <- i+1 to n-1
    dx <- X[i]-X[j]     // compute squared distance
    dy <- Y[i]-Y[j]
    ds <- dx*dx + dy*dy
    if ds < dsmin then
      dsmin <- ds       // new minimum
      im <- i           // memorize point indexes
      jm <- j
    end
  end
end
return (im, jm)  // return closest pair indexes
```

The program is correct because it always terminates (the runtime is in fact $\Theta(n^2)$, as we have seen twice already when we looked at uniqueness and selection sort, which featured the same for loops) and the algorithm looks at all pairs (i, j) with $i < j$, computes squared distances between points i and j, and updates minimum indexes `im,jm` when shorter distances are encountered. The first encountered squared distance will trigger an update, because `dsmin` is initialized with infinity.

This $\Theta(n^2)$ time algorithm is not practical for large point sets, and there exists a $\Theta(n \log n)$ time closest point algorithm that is described in the text books.

The Knapsack Problem

Lecture 12

Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of items that fit into the knapsack.

Example:

```
Item  Weight  Value
  1     7      42
  2     3      12       W = 10
  3     4      40
  4     5      25
```

What is a straightforward approach to solve this problem? Exhaustive search, i.e. enumerating all possible subsets and evaluating packings with regard to their value if they are feasible.


```
Subset    Weight  Value
 {}         0       0
 {1}        7      42
 {2}        3      12
 {3}        4      40
 {4}        5      25
 {1,2}     10      54
 {1,3}     11      -
 {1,4}     12      -
 {2,3}      7      52
 {2,4}      8      37
 {3,4}      9      65   optimal
 {1,2,3}   14      -
 {1,2,4}   15      -
 {1,3,4}   16      -
 {2,3,4}   12      -
 {1,2,3,4} 19      -      - : infeasible
```

The runtime of solving the knapsack problem by exhaustive enumeration is $\Omega(2^n)$, because there are 2^n subsets to consider for n items. For the knapsack problem and a large class of similar combinatorial optimization problems we currently do not know any polynomial time algorithms. Exercise: write a program that solves small knapsack problem instances.

The Job Assignment Problem

Given n workers, n jobs, and a function $c(w, j)$ that specifies the cost of worker w executing job j, find a cost-minimal assignment of all jobs to workers, so that each worker has a job.

Example: cost functions can be represented as matrices, e.g.

c	$j1$	$j2$	$j3$	$j4$
$w1$	9	2	7	8
$w2$	6	4	3	7
$w3$	5	8	1	8
$w4$	7	6	9	4

$c(w1, j2) = 2,\ c(w2, j4) = 8$

Representing both workers and jobs by natural numbers between $1..n$, we are asked to specify a vector of jobs $(j_1, \ldots, j_n)$ so that job j_k is assigned to worker k, each job $1..n$ occurs in the vector exactly once, and its cost

$$\sum_{k=1}^{n} c(k, j_k)$$

is minimal.

Trying to gain some intuition what it takes to solve this problem, we soon realize that for instance the minimal cost entry is not necessarily part of an optimal solution, and more generally no simple strategy seems to work.

So we are tempted to again follow the "brute force" design idea of enumerating all feasible choices and minimizing the cost.

How many feasible assignments are there?

Each job has to be assigned to exactly one worker. This means, that feasible vectors are permutations of $1..n$.

Enumerating all feasible job assignments and computing their cost therefore has runtime $\Omega(n!)$. This means that our brute force algorithm is practically useless even for small n (e.g. $15! = 1,307,674,368,000$).

Fortunately, there is a much more efficient algorithm for this problem called the **Hungarian Method**, whose runtime is $O(n^3)$.