Brute Force + Exhaustive Search

3.1 - 3.3, 3.5

Brute force explained simply

Brute Force Algorithms are exactly what they sound like – straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.

Some examples of brute force

Imagine you have a small padlock with 4 digits, each from 0-9. You forgot your combination, but you don't want to buy another padlock. Since you can't remember any of the digits, you have to use a brute force method to open the lock.

Some examples of brute force (exhaustive)

A classic example in computer science is the traveling salesman problem (TSP). Suppose a salesman needs to visit 10 cities across the country. How does one determine the order in which those cities should be visited such that the total distance traveled is minimized?

The brute force solution is simply to calculate the total distance for every possible route and then select the shortest one. This is not particularly efficient because it is possible to eliminate possible routes through more clever algorithms.

Traveling salesman problem

Let’s watch a video for TSP:
https://www.youtube.com/watch?v=7bpfXh8u6Uk

Performance of brute force algorithms

You can improve a brute force algorithm by reducing the search criteria, or making the problem smaller by not needing to cycle through all options

For example, you have a challenge of finding all the integers between 1 and 100,000,000 that are divisible by 511. If you take a simple approach, Brute Force algorithm would generate all the integers that are in the range. You can reduce the search criteria and make it more efficient by starting with 511 and repeatedly adding the same number until the number exceeds the given limit. Thus, you can save the possible trials by performing analysis in some cases.

How to make a brute force algorithm

Brute Force can be applied to a wide variety of problems. It is used for trial and error problems, searching a number, performing some sort of sorting on the given input unsorted lists, find the integers between some ranges given any condition, find out the largest number and so on. It is extremely useful in solving small-size problems.

Example: sorting

You have a list of n items and you need to arrange them in ascending order. This straightforward problem can be solved by using either selection sort algorithm or bubble sort.

In selection sort algorithm, the program will scan the entire list to find its smallest element and put that smallest element in its final position. Then second step starts with scanning the second element to find the smallest among the n-1 elements. The sorted list will look like this

X0 < X1 < X2 < X3 < X4 < …….. < Xi -1 Xi, ……. , Xn-1

Algorithm: sorting

Problem: To sort a given array X by using selection sort algorithm

Selection Sort (X [0, 1, 2 …… n – 1])

Input: An ordered list of array X [0, 1, 2 …… n – 1] in **descending** order

Output: An ordered list of array X [0, 1, 2…… n – 1] in **ascending** order

Algorithm: sorting

Take a moment to write some brief steps for an algorithm

Simple pseudocode

1. Find the smallest card. Swap it with the first card.
2. Find the second-smallest card. Swap it with the second card.
3. Find the third-smallest card. Swap it with the third card.
4. Repeat finding the next-smallest card, and swapping it into the correct position until the array is sorted.

Simple pseudocode #2

1. Get the value of n which is the total size of the array
2. Partition the list into sorted and unsorted sections. The sorted section is initially empty while the unsorted section contains the entire list
3. Pick the minimum value from the unpartitioned section and placed it into the sorted section.
4. Repeat the process (n – 1) times until all of the elements in the list have been sorted.

Algorithm: sorting

For a <- 0 to n – 2 do

Min <- a

For b <- a + 1 to n – 1 do

If X [b] < X [min] min <- b

Swap X [a] and X [min]

Algorithm: sorting example

You have a list X having elements 84, 43, 66, 94, 28, 31, 11. The following iterations are made on the list

84 43 66 94 28 31 11

11 43 66 94 28 31 84

11 28 66 94 43 31 84

11 28 31 94 43 66 84

11 28 31 43 94 66 84

11 28 31 43 66 94 84

11 28 31 43 66 84 94

Algorithm: sorting example w cards

Real demonstration of algorithm implementation

Algorithm: sorting code

I have attempted to write some Python code as well as C++ code

Play around with it for a moment. Try it with different arrays

Python code [SelectionSortClassEx.py] explained

1. Defines a function named selectionSort
2. Gets the total number of elements in the list. We need this to determine the number of passes to be made when comparing values.
3. Outer loop. Uses the loop to iterate through the values of the list. The number of iterations is (n – 1). The value of n is 5, so (5 – 1) gives us 4. This means the outer iterations will be performed 4 times. In each iteration, the value of the variable i is assigned to the variable minValueIndex
4. Inner loop. Uses the loop to compare the leftmost value to the other values on the right-hand side. However, the value for j does not start at index 0. It starts at (i + 1). This excludes the values that have already been sorted so that we focus on items that have not yet been sorted.
5. Finds the minimum value in the unsorted list and places it in its proper position
6. Updates the value of minValueIndex when the swapping condition is true
7. Compares the values of index numbers minValueIndex and i to see if they are not equal
8. The leftmost value is stored in a temporal variable
9. The lower value from the right-hand side takes the position first position
10. The value that was stored in the temporal value is stored in the position that was previously held by the minimum value
11. Returns the sorted list as the function result
12. Creates a list el that has random numbers
13. Print the sorted list after calling the selection Sort function passing in el as the parameter.

Time complexity of code

We have some working code for a selection sort algorithm

What is the asymptotic run time?

Asymptotic running-time analysis for selection sort

The total running time for selection sort has three parts:

1. The running time for all the calls to *minValueIndex*.
2. The running time for all the calls to *swap*.
3. The running time for the rest of the loop in the *selectionSort* function.

What do you think for #2 and #3?

Asymptotic running-time analysis for selection sort

Parts 2 and 3 are easier. We know that there are n calls to *swap*, and each call takes constant time. Using our asymptotic notation, the time for all calls to *swap* is Θ(n). The rest of the loop in *selectionSort* is really just testing and incrementing the loop variable and calling *minValueIndex* and *swap*, and so that takes constant time for each of the n iterations, for Θ(n) time.

Asymptotic running-time analysis for selection sort

For part 1, the running time for all the calls to *minValueIndex*, we've already done the hard part. Each individual iteration of the loop in *minValueIndex* takes constant time. The number of iterations of this loop is n in the first call, then n-1,, then n-2 , and so on. We've seen that this sum, $1+2+\cdots+(n-1)+n$ is an arithmetic series, and it evaluates to $(n+1)(n/2)$, or $n^2/2 + n/2$. Therefore, the total time for all calls to *minValueIndex* is some constant times $n^2/2 + n/2$. In terms of big-Θ notation, we don't care about that constant factor, nor do we care about the factor of 1/2 or the low-order term. The result is that the running time for all the calls to *minValueIndex* is $\Theta(n^2)$.

$n^2/2 + n/2$ explained

For example, let's say the whole array is of size 8 and think about how selection sort works.

1. In the first call of *minValueIndex*, it has to look at every value in the array, and so the loop body in *minValueIndex* runs 8 times.
2. In the second call of *minValueIndex*, it has to look at every value in the subarray from indices 1 to 7, and so the loop body in *minValueIndex* runs 7 times.
3. In the third call, it looks at the subarray from indices 2 to 7; the loop body runs 6 times.
4. In the fourth call, it looks at the subarray from indices 3 to 7; the loop body runs 5 times.

…

8. In the eighth and final call of *minValueIndex*, the loop body runs just 1 time.

$n^2/2 + n/2$ explained

How do you compute the sum 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 quickly? Here's a trick. Let's add the numbers in a sneaky order. First, let's add 8 + 1, the largest and smallest values. We get 9. Then, let's add 7 + 2, the second-largest and second-smallest values. Interesting, we get 9 again. How about 6 + 3? Also 9. Finally, 5 + 4. Once again, 9! So what do we have?

$$
\begin{aligned}
(8+1)+(7+2)+(6+3)+(5+4) &= 9+9+9+9 \\
&= 4 \cdot 9 \\
&= 36
\end{aligned}
$$

$n^2/2 + n/2$ explained

There were four pairs of numbers, each of which added up to 9. So here's the general trick to sum up any sequence of consecutive integers:

1. Add the smallest and the largest number.
2. Multiply by the number of pairs.

$n^2/2 + n/2$ explained

There were four pairs of numbers, each of which added up to 9. So here's the general trick to sum up any sequence of consecutive integers:

1. Add the smallest and the largest number.
2. Multiply by the number of pairs.

What if the number of integers in the sequence is odd, so that you cannot pair them all up? It doesn't matter! Just count the unpaired number in the middle of the sequence as half a pair. For example, let's sum up 1 + 2 + 3 + 4 + 5. We have two full pairs (1 + 5 and 2 + 4, each summing to 6) and one "half pair" (3, which is half of 6), giving a total of 2.5 pairs. We multiply $2.5 \cdot 6 = 15$, and we get the right answer.

$n^2/2 + n/2$ explained

What if the sequence to sum up goes from 1 to n? We call this an arithmetic series. The sum of the smallest and largest numbers is n+1 . Because there are n numbers altogether, there are n/2 pairs (whether n is odd or even). Therefore, the sum of numbers from 1 to n is (n + 1)(n / 2), which equals $n^2/2 + n/2$.

Asymptotic running-time analysis for selection sort

Adding up the running times for the three parts, we have $\Theta(n^2)$ for the calls to *minValueIndex*, $\Theta(n)$ for the calls to swap, and $\Theta(n)$ for the rest of the loop in *selectionSort*. The $\Theta(n^2)$ term is the most significant, and so we say that the running time of selection sort is $\Theta(n^2)$.

Notice also that no case is particularly good or particularly bad for selection sort. The loop in *minValueIndex* will always make $n^2/2 + n/2$ iterations, regardless of the input. Therefore, we can say that selection sort runs in $\Theta(n^2)$ time in all cases.

Asymptotic running-time analysis for selection sort

Let's see how the $\Theta(n^2)$ running time affects the actual execution time. Let's say that selection sort takes approximately $n^2/10^6$ seconds to sort n values.

Let's start with a fairly small value of n, let's say n = 100.
Then the running time of selection sort is about $100^2/10^2 = 1/100$ seconds.

That seems pretty fast.

Asymptotic running-time analysis for selection sort

Let's see how the $\Theta(n^2)$ running time affects the actual execution time. Let's say that selection sort takes approximately $n^2/10^6$ seconds to sort n values.

But what if n = 1000?
Then selection sort takes about $1000^2/10^6 = 1$ second.

The array grew by a factor of 10, but the running time increased 100 times.

Asymptotic running-time analysis for selection sort

Let's see how the $\Theta(n^2)$ running time affects the actual execution time. Let's say that selection sort takes approximately $n^2/10^6$ seconds to sort n values.

What if n=1,000,000?
Then selection sort takes $1{,}000{,}000^2/10^6 = 1{,}000{,}000$ seconds, which is a little more than 11.5 days.

Increasing the array size by a factor of 1000 increases the running time a million times!

Real life application

Situations when you want to use it include the following:

- You need a sort algorithm that is easy to program (or that requires a small amount of code)

- You only have a small number of elements to sort, so you feel that it is quick enough

- Swaps are expensive on your hardware, but you don't want to use the more complicated Cycle sort.

- You need the sorting time to be consistent for a given size.

What do I mean by swaps being more expensive?

Swap should be more expensive because it includes:

1. Reading data from memory to cache
2. Reading data from cache to registers
3. Writing data back to cache

Swaps vs Comparison

Comparison should be less expensive because it includes:

1. Reading data from memory to cache
2. Reading data from cache to registers
3. Executing single compare operations on two registers (which should be a little faster than writing two integers into a cache)

But modern processors are complex and different from each other, so the best way to get the right answer is to benchmark your code.

Benefits of selection sort

- It's very simple. So, it is easy to program.

- It only requires n swaps (which is better than most sorting algorithms)

- For the same set of elements, it will take the same amount of time regardless of how they are arranged. This can be good for real time applications.

- It performs well on small inputs. (If n is small it will beat O(n log n) sorts)

Cons of selections sort

Here are the Cons:

- $O(n^2)$ is slower than O(n log n) algorithms (like merge sort) for large inputs.

- Insertion sort, which is also $O(n^2)$, is usually faster than it on small inputs.

Group work + discussion

Chapter 3, exercises 5 + 6

A piece of advice for your interviews…

Always go for the brute force method. Even a bad/wildly inefficient solution is better than no solution

Brute force solution gives you a jumping off point that you can optimize from. Once you have a brute force solution, you can use different techniques to improve your time and/or space complexity

The book talks about the “knapsack” problem

https://www.youtube.com/watch?v=YRBON9sIZ2Y