Algorithms – Searching, Greedy Algorithms

**SEARCHING ALGORITHMS**

In computer science, a search algorithm is an algorithm that retrieves information stored within some data structure. Here we are going to look at linear search and binary search.

**Linear Search**

The name itself says that we are going to go in a linear fashion. This algorithm is also called **sequential search.**

It sequentially checks each element of the list for a target value until a match is found or until all the elements have been searched.

*It works on both sorted and unsorted data.*

**Problem Statement:** Given a list L of n elements with values or records and target value T, the following subroutine uses linear search to find the index of the target T in L.

**Basic algorithm:**

1. Set i to 0
2. If Li = T, the search terminates successfully; return i
3. Increase i by 1
4. If i < n, go to step 2. Otherwise, the search terminates unsuccessfully

**Time complexity**

1. **Approach 1**
   1. **Best case:** number of iterations = 1
   2. **Worst case:** number of iterations = n+1 (we didn’t find the element at all)
   3. **Average case :** number of iterations = best + worst / 2 = O(n+2/2) = O(n)
2. **Approach 2: Using recurrence relation**

T(n) = T(n-1) + 1 if n > 1

T(1) = 1

Then by back substitution T(n) = O(n)

**Implementation of linear search on a linked list or array is going to give the same worst case time which is O(n)**

Let’s see the next searching algorithm called Binary Search

**Binary Search**

*Binary search can be only applied when:*

1. *The data is stored in an array*
2. *The data is sorted*

*Binary search cannot be applied when:*

1. *The data is stored in a linked list and sorted*
2. *The data is unsorted*

Binary search, also known as **half-interval search** or **logarithmic search,** is a search algorithm that finds the position of a target value within a sorted array. It does not work on unsorted array.

* Binary search works on the sorted arrays, it begins by comparing the middle element of the array to the target value
* If target value matches the middle element, its position in the array is returned
* If the target value is less than the middle element, then, the search is continued on the left half of the array otherwise the search is continued on the right half of the array

**Problem Statement:** Given an array A of n elements with values or records , sorted such that , and target value T, the following subroutine uses binary search to find the index of T in A.

**Algorithm:**

1. Set L to 0 and R to n-1
2. If L > R, the search terminates as unsuccessful
3. Set m (the position of the middle element) to the floor (the largest previous integer) of (L+R)/2
4. If T = Am, the search is done; return m
5. If T > Am set L to m+1 and go to step 2
6. If T < Am set R to m-1 and go to step 2

**Time complexity:**

Recurrence relation: T(n) = T(n/2) + 1 if n > 1

T(n) = 1 if n = 1

* **T(n) = O(log2n)**

**GREEDY ALGORITHMS**

There are problems where we need to optimize some property like; “minimum cost”, “minimum spanning tree”, “maximize profit”, “maximize the reliability”, “minimize risk” etc. Where we want to maximize or minimize, basically we say that these problems are optimization problems.

There are various ways to solve such problems. Let’s understand with an example. Let’s say I have a graph like below. And we want to find out the smallest path from A to C.



One way to do this is to use **exhaustive search** which means find out all the paths possible and select the minimum from those paths. This is going to be an exponential time or O(2n) algorithm and we don’t want that. We want to see if we can do something better than this.

To solve such optimization problems, we found that there are two programming paradigms.



Using greedy method, we will not be able to solve all the optimization problems. The problem should have some properties then only we can apply greedy method.

We will be able to solve all the optimization problems using dynamic programming, but some programs even after applying DP will give result in exponential time. Therefore, we use dynamic programming where we can get result in polynomial time otherwise it is as good as applying exhaustive search.



Let’s see the problems for which greedy algorithms and dynamic programming can be applied to understand them better.

**KNAPSACK PROBLEM (Greedy)**

Knapsack means a bag. The problem statement is like this.

**Problem Statement:** We are given a bag that has a holding capacity of 20 units. We are given three objects, ob1, ob2 and ob3. Every object has got some profit provided that you sell respective units of that object. What are the objects you will place in the knapsack to get the maximum profit?

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Ob1** | **Ob2** | **Ob3** |
| **Profit** | 25 | 24 | 15 |
| **Weight** | 18 | 15 | 10 |

M (capacity of the bag) = 20 units

1. If we try to put all the objects in the bag, we can’t clearly because the maximum capacity is 20.
2. So what I can do next is that I will try to be ***greedy about profits.*** In this way I will try to add those objects first which will give me the maximum profits and move to the next maximum profit making object. Object 1 is giving me the maximum profit after that object 2 is giving me maximum profit, so I am taking all the Object 1 and will fill the remaining space in the knapsack with the object2.

|  |  |  |
| --- | --- | --- |
|  | Weights | Profits |
| Object 1 | 18 | 25 |
| Object 2 | 2 | 24\*2/15 |
| **Total** | **20** | **28.2** |

But, is this the maximum profit? To answer this, we can go to the next stage.

1. I can now be ***greedy about weights***. Here I will add those objects first which have the least weight and then go to the next object which has the second least weight.

|  |  |  |
| --- | --- | --- |
|  | Weights | Profits |
| Object 3 | 10 | 15 |
| Object 2 | 10 | 24 \* 10/15 |
| **Total** | **20** | **31** |

I found that if I am being greedy about weights, I am getting more profit. But is this the maximum profit again?

1. Next I can go with ***how much profit I am getting per object/weight.*** This way I can go about the **Profit/Weight ratio**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Ob1** | **Ob2** | **Ob3** |
| **Profit** | 25 | 24 | 15 |
| **Weight** | 18 | 15 | 10 |
| **Profit/Weight** | 25/18 = 1.4 | 24/15 = 1.6 | 15/10 = 1.5 |

Intuitively, I will put Object 2 first because I am getting maximum profit per object in this category.

|  |  |  |
| --- | --- | --- |
|  | Weights | Profits |
| Object 2 | 15 | 15\*1.6 = 24 |
| Object 3 | 5 | 5 \* 1.5 = 7.5 |
| **Total** | **20** | **31.5** |

So I found out that neither by being greedy about profit or weights I am getting the maximum profit. I get the maximum profit by using the per weight profit of all the objects and then putting them in the decreasing order of profit.

Let’s see the algorithm and analyze it.

**Greedy knapsack algorithm**

Greedy Knapsack

{

For i = 1 to n:  
 compute pi/wi

Sort objects in non-increasing order of p/w

for i=1 to n from sorted list  
 if(m > 0 && wi <=m)  
 m = m – wi  
 P = P + pi  
 else  
 break;

if (m>0)  
 P = P + pi(m/wi­)

}



Time complexity: T(n) = O(n) + O(n) + O(nlogn) + O(1)

**T(n) = O(nlogn) (worst case time)**

I can also use heap in this problem but **for the worst case** the time complexity will still be O(nlogn).

For using heap sort, we will first make the max heap which takes O(n) and then delete or extract the max from the heap which takes O(logn) time. And this deletion in **worst case** may happen n times.

One can argue that after deletion the depth of the tree may reduce. Yes, that can indeed be the case in which the time complexity may be O(nlog(n-k)) but since we are talking about the **worst case time complexity,** the max heap may be a complete binary tree or almost complete binary tree with (n/2) leaf elements.

Therefore, the complexity = O(n/2 log(n)) which is still **O(nlogn)**

**Example for knapsack:**

Question: Find the maximum profit.

M = 15, N = 5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Objects | 1 | 2 | 3 | 4 | 5 |
| Profit | 2 | 28 | 25 | 18 | 9 |
| Weight | 1 | 4 | 5 | 3 | 3 |

The p/w is

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Object | 1 | 2 | 3 | 4 | 5 |
| Profit/weight | 2 | 7 | 5 | 6 | 3 |

Objects sorted by profit/weight = {2, 4, 3, 5, 1}

Now, M = 15

Knapsack = all object2 + all object4 + all object3 + all object5

Profit = 28 + 18 + 25 + 9 = 80

**Special case of knapsack:** If all of the weights are same, we need not sort them by profit/weight ratio. We can sort them by the profits then and that will be enough

*Let’s move on to our next greedy method which is called Huffman Coding.*