Algorithms & Datastructures II

Proposal 1 – Linear Search

Pseudocode

When a request is made, Linear Search searches through every position in the array A using a *'for loop'* – for element "0", which signals an unused memory position.

If it hits element "0", it will increment *"EmptyBlock"* by 1 for every unused memory position after it until an occupied block is hit.

If *"EmptyBlock"* fulfills the request *"x"*, the first index of the requested block is returned. If not, the for loop will run its whole course and return -1 at the end.

```
A: 1D array                                                      Linear Search
N: Number of elements of array A [memory positions available]
x: Number  ---------------------> [memory positions requested]
Function LinearSearch(A, N, x)
        EmptyBlock <- 0
        for 0 <= i < N do
                if (A[i] == 0) then ------------> //we hit an unused memory position
                        EmptyBlock <- EmptyBlock + 1  //counts unused memory positions
                        if (EmptyBlock == x) then ---> //we have the requested block
                                return i – x + 1 ---> //first index of requested block
                                break
                        end if
                end if
                else EmptyBlock <- 0 [reset]
        end for
        return -1 //we do not have the requested block
end function
```

Running Time, Growth Functions, Theta Notations

Best Case: The best case for the running time of Linear Search occurs when the number (or block) to look for is in the first position checked by the algorithm.

```
Function LinearSearch(A, N, x)
        EmptyBlock <- 0  ------------------------------> C0 (constant value)
        for 0 <= i < N do  ----------------------------> C1 (constant value)
```

```
            if (A[i] == 0) then                         C2 (constant value)
                EmptyBlock <- EmptyBlock + 1            C3 (constant value)
                if (EmptyBlock == x) then               C4 (constant value)
                    return i - x + 1                    C5 (constant value)
end function
```

C1 : For loop executes once in the best case; [number is found in the first check]

C2, C4 : If statements are checked once in the best case.

Running Time

C = [C0 + C1 + C2 + C3 + C4 + C5]

Best case Running time → T(N) = C

Growth Function

Running time → *T(N) = (1)C*

Best case Growth function → 1 [removed coefficients]

Theta Notation

Lower-bound → $c_1 * g(N) \geq T(N)$ for all $N \geq n_0$; where $c_1 = 1$, $g(N) = 1$ and $n_0 = 1$

Upper-bound → $c_2 * g(N) \leq T(N)$ for all $N \geq n_0$; where $c_2 = 2$, $g(N) = 1$ and $n_0 = 1$

Finally, $c_1 * (1) \leq T(N) \leq c_2 * (1)$ for all $N \geq n_0$; where $c_1 = 1$, $c_2 = 2$, $g(N) = 1$ and $n_0 = 1$

Best case Theta notation → T(N) = Θ(1)

Worst Case: The worst case for the running time of Linear Search occurs when the number (or block) to look for is in the last position checked or it is not in the array.

```
Function LinearSearch(A, N, x)
    EmptyBlock <- 0                                     C0 (constant value)
    for 0 <= i < N do                                   C1 * N + C2
        if (A[i] == 0) then                             C3 * N
            EmptyBlock <- EmptyBlock + 1                C4 (constant value)
            if (EmptyBlock == x) then                   C5 * N
                return i – x + 1
        else EmptyBlock <- 0                            C6 (constant value)
    return -1                                           C7 (constant value)
end function
```

C1, C2 : For loop executes N times + 1 considering the false condition at the end of the loop.

C3, C5 : If statements execute N times.

Running Time

$C8 = [C0 + C2 + C4 + C6 + C7]$

$C9 = [C1 + C3 + C5]$

Worst case Running time → $T(N) = (C1 * N) + (C3 * N) + (C5 * N) + C8$

→ $C9 * N + C8$

Growth Function

Running time → $T(N) = C1 * N + C8$

Worst case Growth function → N [removed coefficients]

Theta Notation

Lower-bound → $c_1 * g(N) \geq T(N)$ *for all* $N \geq n_0$ *; where* $c_1 = 1$, $g(N) = N$ *and* $n_0 = 1$

Upper-bound → $c_2 * g(N) \leq T(N)$ *for all* $N \geq n_0$ *; where* $c_2 = 10$, $g(N) = N$ *and* $n_0 = 1$

Finally, $c_1 * (1) \leq T(N) \leq c_2 * (1)$ *for all* $N \geq n_0$ *; where* $c_1 = 1$, $c_2 = 10$, $g(N) = N$ *and* $n_0 = 1$

Worst case Theta notation → $T(N) = \Theta(N)$

Proposal 2 – Direct Search

Pseudocode

When a request is made, Direct Search checks the condition for the program to execute – Is there enough memory allocated to a chunk? (N/M) If not, it will reject the request immediately.

It then checks array POS for any space to allocate the requested memory (in array A) – Any number not -1 means there is space, and -1 means there is no space.

The program returns POS[x - 1] if the request is accepted. If not, it returns -1.

```
A: 1D array                                                    Direct Search

N: Number of elements of array A

x: Number ——————> [memory positions requested]

POS: 1D array ———> [contains M elements]

M: Number of chunks [maximum requestable]

Function DirectSearch(A, N, x, POS, M)

       chunkSize <- N/M

       if (chunkSize > M) then ————> //enough memory per chunk
```

```
            value <- POS[x – 1]  ──▶ //checks array POS for space
            if (value != -1) then ──▶ //space is not occupied
                  allocate() ──▶ //allocates the requested blocks in array A
                  return value [request accepted]
            end if
      end if
      else return -1 [request rejected]
end function
```

B – E) Running Time, Growth Functions, Theta Notations

Best Case: The best case for the running time of Direct Search occurs when the chunk size (N/M) is smaller than M. When this happens, it means the condition to run the rest of the algorithm is false.

```
Function DirectSearch(A, N, x, POS, M)
      chunkSize <- N/M  ──────────▶ C0 (constant value)
      if (chunkSize > M) then ──────▶ C1 (constant value)
      else return -1  ────────────▶ C2 (constant value)
end function
```

C0 : Assigning values takes constant time.

C1 : If statement checks the condition once, which executes in constant time.

C2 : Returning values takes constant time.

Running Time

C = [C0 + C1 + C2]

Best case Running time → T(N) = C

Growth Function

Running time → *T(N) = (1)C*

Best case Growth function → 1 [removed coefficients]

Theta Notation

Lower-bound → $c_1 * g(N) \geq T(N)$ *for all* $N \geq n_0$ *; where* $c_1 = 1$*,* $g(N) = 1$ *and* $n_0 = 1$

Upper-bound → $c_2 * g(N) \leq T(N)$ *for all* $N \geq n_0$ *; where* $c_2 = 2$*,* $g(N) = 1$ *and* $n_0 = 1$

Finally, $c_1 * (1) \leq T(N) \leq c_2 * (1)$ *for all* $N \geq n_0$ *; where* $c_1 = 1$*,* $c_2 = 2$*,* $g(N) = 1$ *and* $n_0 = 1$

Best case Theta notation → T(N) = Θ(1)

Worst Case: The worst case for the running time of Direct Search occurs when the number (or block) to look for is in the position checked.

```
Function DirectSearch(A, N, x, POS, M)
        chunkSize <- N/M                  → C0 (constant value)
        if (chunkSize > M) then           → C1 (constant value)
                value <- POS[x – 1]       → C2 (constant value)
                if (value != -1) then     → C3 (constant value)
                        allocate()        → C4 (constant value)
                        return value      → C5 (constant value)
end function
```

C0, C2 : Assigning values takes constant time.

C1, C3 : If statement checks the condition once, which executes in constant time.

C4 : Calling the function *allocate* takes constant time.

C5 : Returning values takes constant time.

Running Time

C = [C0 + C1 + C2 + C3 + C4 + C5]

Worst case Running time → T(N) = C

Growth Function

Running time → *T(N) = (1)C*

Worst case Growth function → 1 [removed coefficients]

Theta Notation

Lower-bound → $c_1 * g(N) \geq T(N)$ *for all* $N \geq n_0$ *; where* $c_1 = 1$, $g(N) = 1$ *and* $n_0 = 1$

Upper-bound → $c_2 * g(N) \leq T(N)$ *for all* $N \geq n_0$ *; where* $c_2 = 2$, $g(N) = 1$ *and* $n_0 = 1$

Finally, $c_1 * (1) \leq T(N) \leq c_2 * (1)$ *for all* $N \geq n_0$ *; where* $c_1 = 1$, $c_2 = 2$, $g(N) = 1$ *and* $n_0 = 1$

Worst case Theta notation → T(N) = Θ(1)

Proposal 3 – Exhaustive Search

Pseudocode

When a request is made, Exhaustive Search searches through every position in the array A using a *'for loop'* – for element "0", which signals an unused memory position. This is similar to *Linear Search.*

If it hits element "0", it will increment *"EmptyBlock"* by 1 for every unused memory position after it until an occupied block is hit.

We check if EmptyBlock can fulfill the request x. **(EmptyBlock >= x)** If it can:

Two values are then calculated using the current EmptyBlock; current value and current index.

→ Current value is derived by A – x. The lower the value, the better the chunk fits the request.

→ Current index is the first index of the chunk of empty blocks, derived by i – x.

Next, we compare current value against lowest value. **(current < lowest)**

→ For the first chunk, (current < lowest) will always be true as lowest is initalized as infinity (∞).

After the loop has gone through the whole array A (N times), the **lowest index** will be returned.

```
A: 1D array                                                    Exhaustive Search
N: Number of elements of array A [memory positions available]
x: Number  ------------------->  [memory positions requested]

Function ExhaustiveSearch(A, N, x)
    lowest <- infinity
    for 0 <= i < N do
        if(A[i] == 0) then ----------------------> //we hit an empty block
            EmptyBlock <- EmptyBlock + 1 -------> //count empty blocks
        end if
        else [end of empty blocks]
            if(EmptyBlock >= x) then -----------> //we can fulfill request
                current <- EmptyBlock – x [value A – x]
                currentIndex <- i – x
                if(current < lowest) ----------> //compare A – x value
                    lowest <- current [best value]
                    lowestIndex <- currentIndex
                end if
            end if
            EmptyBlock <- 0 [reset count]
```

```
                continue
            end else
            return lowestIndex
end function
```

B – E) Running Time, Growth Functions, Theta Notations

Best and Worst Case: The best and worst case for the running time of Exhaustive Search are the same as the algorithm has to search through the whole array to find the best slot to fit the requested memory.

```
Function ExhaustiveSearch(A, N, x)
    lowest <- infinity                      ----> C0 (constant value)
    for 0 <= i < N do                       ----> C1 * N + C2
        if (A[i] == 0) then                 ----> C3 * N
            EmptyBlock <- EmptyBlock + 1    ----> C4 (constant value)
        else
            if (EmptyBlock >= x) then       ----> C5 * N
                current <- EmptyBlock – x   ----> C6 (constant value)
                currentIndex <- i – x       ----> C7 (constant value)
                if (current < lowest)       ----> C8 * N
                    lowest <- current       ----> C9 (constant value)
                    lowestIndex <- currentIndex ----> C10
            EmptyBlock <- 0                 ----> C11 (constant value)
            continue
        return lowestIndex                  ----> C12 (constant value)
end function
```

C1, C2 : For loop executes N times + 1 considering the false condition at the end of the loop.

C3, C5, C8 : If statements execute N times.

Running Time

C13 = [C0 + C2 + C4 + C6 + C7 + C9 + C10 + C11 + C12]

C14 = [C1 + C3 + C5 + C8]

Best and Worst case Running time → ***T(N) = (C1 * N) + (C3 * N) + (C5 * N) + (C8 * N) + C13*** ***→ C14 * N + C13***

Growth Function

Running time → $T(N) = \mathbf{C14 * N + C13}$

Best and Wost case Growth function → N [removed coefficients]

Theta Notation

Lower-bound → $c_1 * g(N) \geq T(N)$ *for all* $N \geq n_0$ *; where* $c_1 = 1$, $g(N) = N$ *and* $n_0 = 1$

Upper-bound → $c_2 * g(N) \leq T(N)$ *for all* $N \geq n_0$ *; where* $c_2 = 10$, $g(N) = N$ *and* $n_0 = 1$

Finally, $c_1 * (1) \leq T(N) \leq c_2 * (1)$ *for all* $N \geq n_0$ *; where* $c_1 = 1$, $c_2 = 10$, $g(N) = N$ *and* $n_0 = 1$

Best and Worst case Theta notation → T(N) = Θ(N)

F) Recommendations

Let's look at all three proposals and discuss the advantages and disadvantages they offer.

Direct Search has the fastest running time, but performs badly in terms of memory fragmentation.

Running Time (Cost)

Direct Search has a running time is ***Θ(1) and Θ(1)*** for best and worst cases respectively.

- This means that the size of the array used does not matter.

It has the lowest cost out of the three proposals.

Memory Fragmentation (Performance)

Direct Search uses chunks to allocate memory. Hence, some requests may not be fulfilled even though there are empty memory positions.

It has the worst performance out of the three proposals.

Other considerations

However, both factors above do not account for the array POS, which is key to the fast running time of Direct Search.

There is an external function for array POS that runs separate from Direct Search which behaves similar to *Exhaustive Search*. To achieve array POS, the algorithm has to search through the whole array A and then return the index of the first available spot for each chunk. This may take a lot of time if the array A has a large size.

Recommendation

Considering all factors above, I can recommend ***Direct Search*** for the scenarios listed below:

1. The content of array A does not have to change frequently.

In this case, it may take some time to compute the results for array POS, but it only has to be done once or a few times. After the initial computation, Direct Search's value will prevail over the others because of it's speed of searching.

2. The size of array A is small.

In this case, it may not take very long to compute the results for array POS, so it may be a worthwile tradeoff between running time and performance. However, this also means that the maximum requestable position has to be small or else there will not be enough space allocated to each chunk.

Exhaustive Search has the slowest running time, but performs well in terms of memory fragmentation.

Running Time (Cost)

Exhaustive Search has a running time is ***Θ(N)*** for both best and worst case.

- This means that the bigger the array, the longer it will take to run the whole algorithm.

It has the highest cost out of the three proposals.

Memory Fragmentation (Performance)

Exhaustive Search searches through the whole array for the best position to allocate memory. Hence, it makes sure that almost all requests are fulfilled.

It has the best performance out of the three proposals.

Recommendation

Considering all factors above, I can recommend ***Exhaustive Search*** for the scenarios listed below:

1. Performance is valued over speed.

In some cases, the cost of running Exhaustive Search may not be proportional to the performance. However, this search will always aim to deliver a solution if it exitsts. Therefore, if performance is priority, then this is the recommended proposal to use.

2. The size of array A is small.

In this case, it may not take long to search through the whole array for the best fit position.

3. Ease of implementation is needed.

Exhaustive Search is easy to implement as it searches for the best solution by exhausting all options, like the name suggests, and without comparing elements.

Linear Search offers a balance between Direct and Exhaustive Search. It performs decently in terms of running time and memory fragmentation but not as good as the two searches in each area.

Running Time (Cost)

Linear Search has a running time of ***Θ(1) and Θ(N)*** for best and worst cases respectively.

- This means that the running time is dependent on the content of the array and the requests made to it.

Memory Fragmentation (Performance)

- Similar to it's running time, Linear Search's performance is dependent on the content of the array and the requests made to it.

Recommendation

1. The algorithm collaborates with other sorting algorithms.

In this example of Linear Search, it stops searching when it hits the first block that can fulfill a request, regardless of whether it is the best solution. Hence, when Linear Search is paired with another sorting algorithm, the cost of running it may improve greatly.

An example of sorting is to arrange the array from empty positions at the start, to filled postitions ar the end.

2. The size of the array is not too big.

Linear Search is most practical to implement with a medium sized array. In this case, there will be most minimal tradeoff between cost and performance.

[Part 2 – Propose a New Algorithm]

A) Description

The idea behind the algorithm is to improve on the existing Linear Search algorithm, by having it push all occupied memory positions together to increase performance, while maintaining the (worst case) running time; having the sorting algorithm run in the same, existing for loop.

As you can see in the figure below, there are 2 chunks of unallocated memory positions. These chunks add up to 5 total positions, but the maximum that it can fulfill is 3 → from index 7 – 9

Before:

1	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**
[0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**

Swap index [4] with index [2] and so on…

Code explanation:

The variable *sortCounter* acts as the position of the earliest unoccupied memory space. Following the figure above, *sortCounter* would have the value 2. When the *for loop counter i* reaches [4], we can swap index [4] and [2] values easily.

After:

The maximum it can fulfill will increase to 5 instead of 3 → from index 5 – 9

1	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**
[0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**

A) Pseudocode

```
A: 1D array                                              Modified Linear Search
N: Number of elements of array A [memory positions available]
x: Number  ------------------>  [memory positions requested]
Function NewSearch(A, N, x)
        sortCounter <- 0 //counter for non empty block (for sorting)
        emptyBlock <- 0
        index <- 0
        for (0 <= i < N)
                if (A[i] != 0) //space is occupied
                        if (sortCounter != i)
                                A[sortCounter] <- A[i]      Swaps positions
                                A[i] <- 0                   (see above)
                                sortCounter <- sortCounter + 1
                        end if
                        /**if sortCounter and i are in
                        the same position, there is no
                        need to swap them**/
                        else
                                A[counter] <- A[i] //occupied block stays the same
                                sortCounter <- sortCounter + 1
                        end else
                end if
                else                                                Linear Search
                        emptyBlock <- emptyBlock + 1
                end else
                if (emptyBlock >= x) //can fulfill request
                        return N – emptyBlock
                end if
                else return -1
                end else
        end for
end function
```

C – F) Running Time, Growth Functions, Theta Notations

Best and Worst Case: The best and worst case for the running time of the Modified Linear Search are the same as the sorting algorithm will have to search through the whole array (N times) in order to sort the whole array first.

The searching algorithm also exists in the same for loop, so they share the running time.

```
Function LinearSearch(A, N, x)
    EmptyBlock <- 0                          → C0 (constant value)
    for 0 <= i < N do                        → C1 * N + C2
        sortingAlgorithm()                   → C3 (constant value)
        if (A[i] == 0) then                  → C4 * N
            EmptyBlock <- EmptyBlock + 1     → C5 (constant value)
            if (EmptyBlock == x) then        → C6 * N
                return i - x + 1             → C7 (constant value)
end function
```

C1, C2 : For loop executes N times + 1 considering the false condition at the end of the loop.

C4, C6 : If statements execute N times as they are in the for loop.

Running Time

C8 = [C0 + C2 + C3 + C4 + C5 + C6 + C7]

*C9 = [C1 * C4 + C6]*

Best and Worst case Running time → T(N) = C9 * N + C8

Growth Function

Running time → *T(N) = C9 * N + C8*

Worst case Growth function → N [removed coefficients]

Theta Notation

Lower-bound → $c_1 * g(N) \geq T(N)$ *for all* $N \geq n_0$ *; where* $c_1 = 1$, $g(N) = N$ *and* $n_0 = 1$

Upper-bound → $c_2 * g(N) \leq T(N)$ *for all* $N \geq n_0$ *; where* $c_2 = 10$, $g(N) = N$ *and* $n_0 = 1$

Finally, $c_1 * (1) \leq T(N) \leq c_2 * (1)$ *for all* $N \geq n_0$ *; where* $c_1 = 1$, $c_2 = 10$, $g(N) = N$ *and* $n_0 = 1$

Worst case Theta notation → $T(N) = \Theta(N)$

G) Reflective writing

This was a tough exercise as all three proposals had running times that were quite low (1 to N), so there weren't many options for running times smaller than N. It could only go lower to NlogN, or 1.

That, or improve performance without increasing running time significantly, or else I would not be able to justify using my “improved” algorithm against the 3 proposed ones.

This improved algorithm builds upon the existing Linear Search as seen in proposal 1. It aims to increase performance by forming all unoccupied memory positions together, so as to be able to take in bigger memory requests.

The only way I could keep the running time more or less similar was to nest the sorting algorithm within the same for loop as the searching one. If I had another for loop running before the original for loop, the running time would be 2N, but the growth function would still be N after removing coefficients. I thought of doing this, but it didn’t seem logical in my opinion.

I believe this improved algorithm can now be used over exhaustive search in more cases than before as they have the same running times but Linear Search has better performance.

I can only show this using the figure below:

1	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**
[0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**

If Exhaustive Search is used on the array above, it would only be able to fill up to a maximum request of 3 memory positions.

If Linear Search is used on the array above, it would be able to fulfill requests of up to 5 memory positions. Hence, Linear Search offers more options.

Thank you for reading my report. I know it was lengthy 😊