ALGORITHMS & DATA STRUCTURES 2
Mid-Term Coursework
Give me space!

1. INTRODUCTION

Every time a program creates a variable, enough memory space for it must be allocated. Depending on the type of variable, a different number of memory positions are allocated to that variable. For example, in Java and C++ an integer variable requires 4 contiguous memory positions whilst a float variable requires 8 contiguous memory positions. If your program creates an array of 10 integers, then 40 contiguous memory positions need to be allocated.

When the execution of your program finishes, the corresponding allocated memory positions are freed so other programs being executed in the machine can use them.

With the continuous allocation/release of memory positions a problem can emerge known as *memory fragmentation*. That is, the available memory blocks are not contiguous, but "interrupted" by used memory blocks. Therefore, a request from a program requiring a block of X contiguous memory positions might be rejected even if there were X available positions because they are not contiguous.

Figure 1 illustrates the problem of memory fragmentation using a contrived example of a memory with 16 memory positions (that would be a very useless memory of 16 bytes!). In this example, Programs 1, 2, 3 and 4 have already requested (and been allocated) 2, 3, 2, and 6 memory positions, respectively (upper part of the figure). Then, Program 2 finishes its execution, releasing its allocated memory positions (central part of the figure). Finally, Program 5 requests 5 memory positions, but the request is rejected even when there are enough available memory positions, because they are not contiguous.

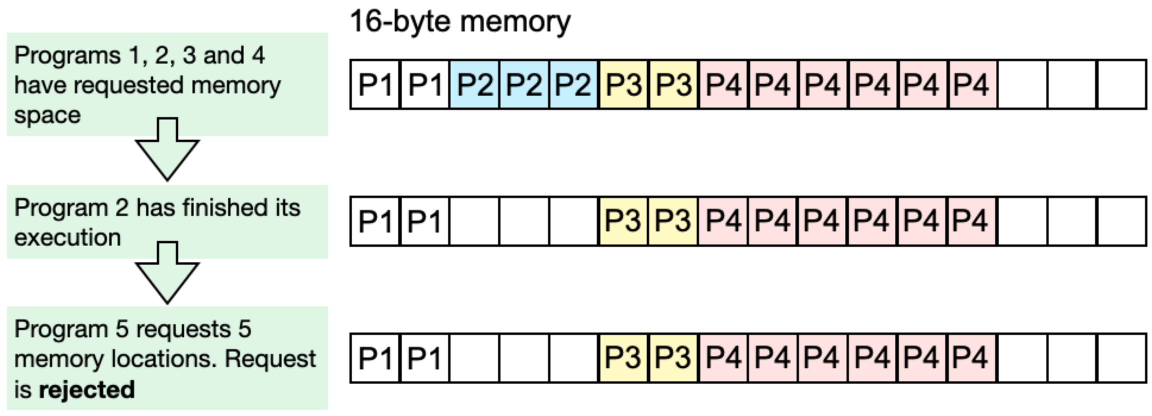
16-byte memory
Programs 1, 2, 3 and 4 have requested memory space
Program 2 has finished its execution
Program 5 requests 5 memory locations. Request is **rejected**
P1 P1 P2 P2 P2 P3 P3 P4 P4 P4 P4 P4 P4
P1 P1 P3 P3 P4 P4 P4 P4 P4 P4
P1 P1 P3 P3 P4 P4 P4 P4 P4 P4

Figure 1. The memory fragmentation problem

The team in charge of implementing an algorithm that searches for free memory positions has received 3 proposals that aim to search for available memory positions in a memory made of N elements. The aim of the algorithms is striking a balance between fast running time and low memory fragmentation. Assume that the arriving requests can ask for a number of memory positions in the range from 1 to M (M << N), both inclusive. The proposals are:

Linear Search: When a request of X memory positions is received, scan the memory positions from left to right. Allocate the first block with X or more available positions. As soon a block is found, return the position of the first element. If not, return -1.

By way of example, assume the 16-byte memory is in the state shown in Figure 2 (this occupation state is the result of many past memory allocation and release processes).

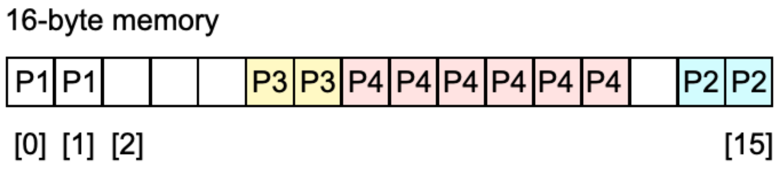

Figure 2. State of a 16-byte memory at a given moment

If a new request for 1 memory position arrives, Linear Search would then find memory position [2] to for it, as this is the first block of at least 1 memory position available when scanning the memory from left to right. Notice that, in terms of fragmentation, selecting memory position [13] would be a better option but this algorithm does not allow that selection.

Notice that the task of updating the state of the memory positions to mark them as 'allocated' is not part of the Linear Search algorithm.

Direct Search: The memory is divided in M chunks, each equally made of N/M memory positions (remember that M is the maximum number of memory positions a program can request). Requests of 1 memory position are allocated in the first chunk; requests of 2 memory positions are allocated in the second chunk and so on. Notice that the chunk size N/M must be greater than M, otherwise a program will not have enough memory allocated to a chunk. To speed up the search of an available block of memory positions in each chunk, the M-element array POS is kept. Assume the variables used to manage memory positions – as the array POS - are stored in a reserved part of the memory, not allocated to programs. The first element of POS stores the position (in the memory) where the first available block of 1 memory position is located, the second element stores the position (in memory) where the first available block of 2 memory positions is located and so on. If the ith element in the array POS stores number -1, it means that there are no blocks of (i+1) memory positions available in the i-th chunk. When a request of X memory positions is received, the algorithm simply checks the element (X-1) in the array POS. If it stores a value different from -1, it means there is space to allocate a block of X. In that case, the value stored in POS[X-1] is returned. Otherwise, the value -1 is returned (request rejected).

By way of example, assume M=4 with different programs requesting 1, 2, 3 or 4 memory positions. In that case, the memory would be divided in 4 chunks. Figure 3 shows a possible state of our 16-byte memory, divided in 4 chunks. In the figure, the content of the array POS is also shown.

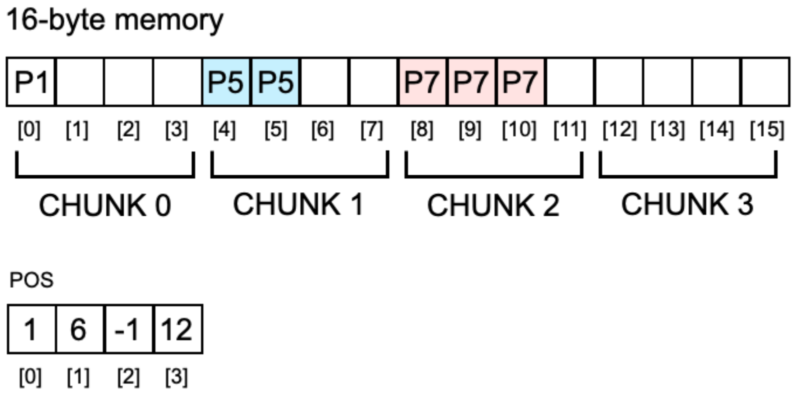

Figure 3. State of a 16-byte memory at a given moment along with array POS

In Figure 3:

- Chunk 0 is used to allocate memory to programs requesting 1 memory position. Thus, there is space to allocate 4 requests of 1 memory position each. In this example, the next available block of 1 memory position in Chunk 0 starts at position [1] in the memory. So, array POS stores the value 1 in position [0] (position [0] corresponds to Chunk 0).
- Chunk 1 is used to allocate memory to programs requesting 2 memory positions. Thus, there is space to allocate 2 requests of 2 memory positions each. In this example, the next available block of 2 memory positions in Chunk 1 starts at position [6] in the memory. So, array POS stores the value 6 in position [1] (position [1] corresponds to Chunk 1).
- Chunk 2 is used to allocate memory to programs requesting 3 memory positions. Thus, there is space to allocate only 1 request of 3 memory positions (and 1 memory position is lost). In this example, there is no next available block of 3 memory positions in Chunk 1 and thus, array POS stores the value -1 in position [2] (position [2] corresponds to Chunk 2).
- Chunk 3 is used to allocate memory to programs requesting 4 memory positions. Thus, there is space to allocate only 1 request of 4 memory positions. In this example, the only available block of 4 memory positions in Chunk 1 starts at position [12] in the memory. So, array POS stores the value 12 in position [3] (position [3] corresponds to Chunk 3).

In this example, if a program requests 2 memory positions, the algorithm checks the value in POS[1]. Because this value is different to -1, it means there is a block of 2 free memory positions starting at position [6] in the memory. So, the value 6 is returned.

Notice that the task of actually allocating the found memory positions to the request and updating the array POS is not performed by Direct Search. Direct Search only finds a suitable block of free memory positions. Thus, another function allocates the memory positions [6] and [7] (marking them as used) and updates the value of POS[1] to -1.

Exhaustive Search: When a request for X memory positions is received, the block of available positions that best fits the request is allocated. That is, the block with the lowest value for (A-X), where A is the number of available positions in that block. If a block is found, return the position of the first element. If not, return -1.

By way of example, consider the same example shown in Figure 2. Assume that a 1-memory position request arrives. In this case, Exhaustive Search would have two possible options: allocating memory position [2], with a value of (A-X) equal to (3-1)=2 or allocating memory position [13], with a value of (A-X) equal to (1-1)=0. Since the lowest value is the one corresponding to position [13], this value (13) is returned by the algorithm.

Notice that the task of updating the state of the memory positions to mark them as 'allocated' is not part of the Exhaustive Search algorithm.

EXECUTION EXAMPLE

Figure 4 shows an example of the execution of these 3 algorithms starting with an empty 8-byte memory. The programs can request 1, 2 or 3 memory positions (M=3). In the case of Direct Search, since the memory size (8 memory positions) is not divided exactly by 3, we will arbitrarily assume that all chunks will have 3 memory positions, except Chunk 1.

Assume the algorithms must process the following sequence of events:

- Program 1 (P1) requests 2 memory positions
- Program 2 (P2) requests 3 memory positions
- Program 3 (P3) requests 1 memory position
- P2 ends its execution (all allocated memory positions to P2 are released)
- Program 4 (P4) requests 2 memory positions
- Program 1 requests 3 memory positions

For this particular sequence of events, you can see that:

- Linear Search rejects the request of Program 1 for 3 memory positions (the last request), in spite of having 3 available memory positions. However, due to memory fragmentation, they were not contiguous.
- Direct Search rejects the request of Program 4 for 2 memory positions (the one before last request), in spite of having enough available contiguous positions in Chunks 0 and 2. Also note that in this specific execution of Direct Search, the middle chunk only

contains two memory locations; this is because we do not have a number of memory locations that is perfectly divisible by three, and assigning three locations for a chunk that only takes two memory units means one location will never be used.

- Exhaustive Search is able to find a suitable block of free memory positions to all requests.

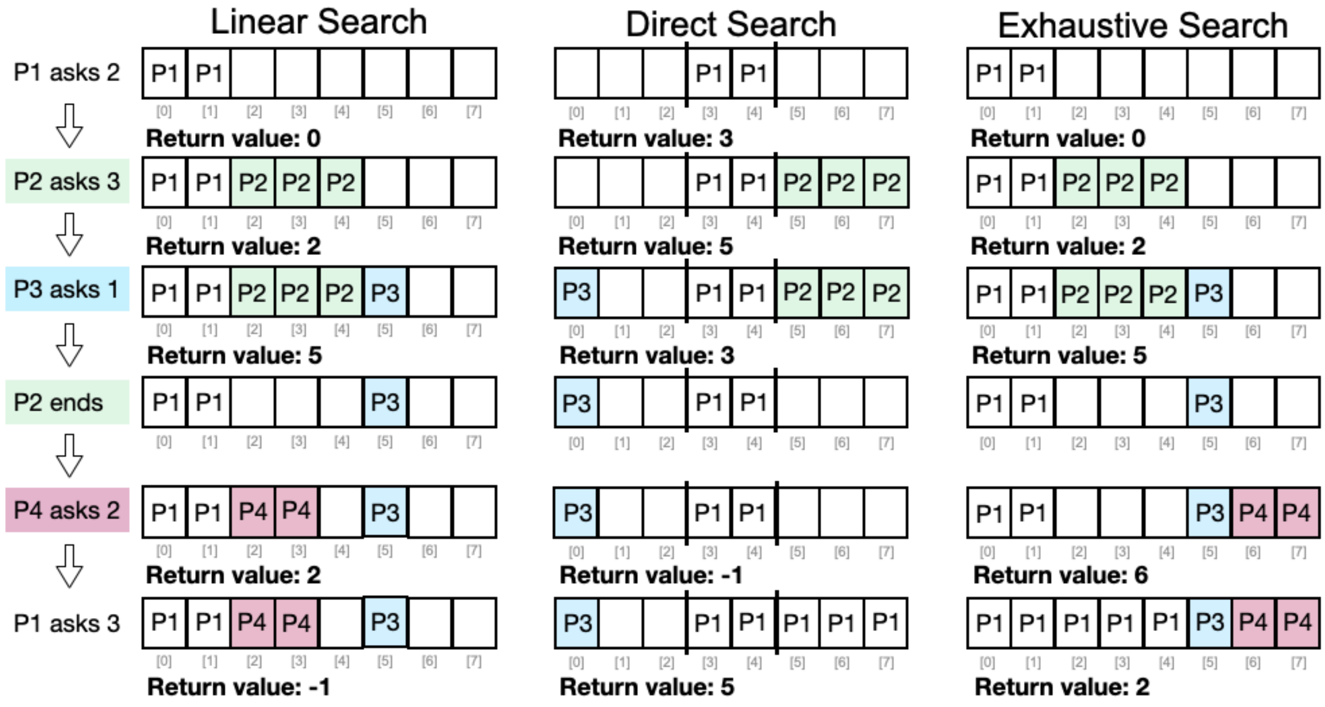

Figure 2. Example of execution of proposals for a particular sequence of events

Notice that the above explanation is a (very) simplified model of the memory allocation process that occurs in a real computer. Those details are out of the scope of this assessment.

2. YOUR TASK

Your mid-term task is divided into two parts:

Part 1: Analyse the 3 algorithmic proposals just described and then recommend the one to be implemented. Your analysis must include:

a) The **pseudocode** description of each proposal. Every proposal receives 3 input arguments: A (array representing the memory), N (the size of the memory) and X (the number of memory positions requested). Direct Search also receives array POS and number of chunks M as input arguments. In our simplified representation of the computer memory, assume that storing a number 0 in a memory position signals a free memory position.
In the case your pseudocode includes functions for which we have well known algorithms already studied in this module you don't need to write the pseudocode for those functions, e.g. binary search, mergesort, counting sort, etc. You can simply call these algorithms in your pseudocode. You may also add comments to your pseudocode to increase its clarity. For the case of Direct Search, you can assume that the number M divides N exactly (all chunks are of exactly the same size). If you need to make any further assumptions, please make them explicit in the document you will submit.

b) The description of the **worst and best cases for the running time** of each algorithm. Describe in detail the situation when the worst and best cases occur. For example, if you had to describe the best case for the running time of the classical Linear Search algorithm (not the one presented here, but the one that looks for a number in an array) you would write *"The best case for the running time of Linear Search occurs when the number to look for is in the first position checked by the algorithm. The worst case for the running time of Linear Search occurs when the number to look for is in the last position checked or it is not in the array"*. Please, notice that we are talking about the worst/best cases in terms of running time (not in terms of fragmentation or any other metric).

c) An **expression for the running time (or execution time), T(N),** for the worst and best case of each proposal. In deriving each expression, make sure you explain how each line of pseudocode translates to a constant value or to an expression that depends on the size of the input data. Just writing the function describing the running time is not enough. You must show you know how to derive it from the pseudocode. For example, to show how to derive the expression for the execution time T(N) of the classical Linear Search (not the one presented here for the memory allocation problem, but the one that looks for a number in an array) you should do something like:

Best case: element found in first attempt

```
function LS(A,N,x)
     for 0 <= i < N       ----> C0 (constant value)
          if(A[i]==x)      ----> C1 (constant value)
               return i    ----> C2 (constant value)
     return -1
end function
```

T(N)= C (C=C0+C1+C2, constant value)

Worst case: element not found in the array

```
function LS(A,N,x)
     for 0 <= i < N       ----> K0*N+K1
          if(A[i]==x)      ----> K2 (constant value)
               return i
     return -1             ----> K3 (constant value)
end function
```

*T(N)= K0*N+K4 (K4=K2+K3)*

d) The **growth functions** of the execution time for each proposal, for worst and best cases. Remember that the growth function is the function that describes how fast a function grows (in this case, the function describing the running time of an algorithm). Thus, a growth function does not include constants or low order terms. In the example above, the growth functions for the best and worst case would be 1 and N, respectively. That is, the best case has a constant running time and the worst case grows with N.

e) The **Theta notation** for the worst and best cases of each proposal. In particular, find a set of values of c_1, c_2 and n_0 for which T(N) is Theta(g(N)). The values c_1, c_2 and n_0 can be actual numbers (if you derived T(N) counting up in detail every time unit) or algebraic (if you used generic constants for the expression of T(N)). For the example of the classical Linear Search algorithm, you should do something similar to the following reasoning:

Best case T(N) is Theta(1) because $c_1*1 \leq T(N) \leq c_2*1$ *for all* $N > n_0$, *for* $c_1=1$ *and* $c_2=2*C$ *and* $n_0=1$.

Worst case T(N) is Theta(N) because $c_1*N \leq T(N) \leq c_2*N$ *for all* $N > n_0$, *for* $c_1=1$ *and* $c_2=(10*K0*K4)$ *and* $n_0=1$.

f) Based on the previous findings, discuss what proposal you would recommend to implement and why. This part is a **reflective writing exercise** where you consider different use cases (e.g. when Linear/Direct/Exhaustive Search is the best/worst choice) and discuss why one or another proposal would be best in those cases. It is not enough to just name one proposal, you must discuss **why** this proposal is best. Maybe there is no a single proposal that is best for all cases. Maybe one is good in terms of fragmentation (increasing the number of requests served), but it is not the fastest one. Make sure you comment at least on: **time complexity and performance** (number of requests accepted). For example, if you were comparing the classical Linear Search algorithm to the Binary Search one you might comment something like this: *"My recommendation depends on the environment where the search algorithms are going to be used. In an environment where the content of the array changes constantly, I recommend to implement Linear Search (N), in spite of it having a higher time complexity than Binary Search (logN). This recommendation stems from the fact that Binary Search requires the array to be sorted before performing the search. The cost of sorting the numbers is not considered in the expression for the time complexity of Binary Search (logN) that only considers the searching part after the array has been sorted. If the array changes constantly and requires permanent sorting, the time complexity of sorting (NlogN if we use Mergesort or N is we can use counting sort) dominates the time complexity of using Binary Search and its advantage over Linear Search is lost. On the contrary, if the array needs to be sorted only once at the beginning and then, only the search part must be performed a significant amount of times, I recommend to implement Binary Search because in this case the time complexity of the algorithm is actually dominated by the search part".* In the case of the memory allocation problem you might also want to comment on how the search algorithms collaborate with other functions required for the whole system to work (e.g. the process of actually allocating the memory positions or the algorithm in charge of releasing resources).

Finally, you might want to check that your pseudocode and your final conclusion is correct by:

- Implementing a functional C++ version of each proposal. This will help you to check that your pseudocode is correct and your complexity analysis is correct. For example, you can run each proposal for big memory sizes with a specific long sequence of requests/releases and check that the fastest algorithm is also the one with the lowest time complexity. Or, you can check whether – for different randomly created sequences of memory requests/releases – the proposal you thought was the best in terms of number of requests accepted is actually the best one.
- Comparing the growth function with the actual execution time of your code, for each proposal. To measure the execution time of your code, simply execute a variant of the time() function at the beginning and at the end of your code. This measurement is not exact and it has all the drawbacks discussed at the beginning of the course, but it allows you to make a quick comparative empirical analysis of the proposals. You can then plot the results: the horizontal axis is the number of rows (columns) of the matrix and the vertical axis is the execution time.

These two last activities (implementing the code and making the empirical comparison) are **not compulsory** and they will not be marked. That is, you do not need to report on this part of your work. You only need to report on parts a)-f). This suggestion is just given to you as a way to check the correctness of your work before submitting and having extra insights on the different parts of the whole system; do not submit any code files, just a pdf.

We encourage you to use C++, as this is the programming language used in the practical exercises of this module. However, as this part is not marked, you can use any programming language you wish.

Part 2: Propose a brand new algorithm, or an algorithm that corresponds to an improvement to any of the 3 proposals just analysed. The improvement can be in terms of running time, memory consumption or performance (more requests accepted). If you propose a brand new algorithm, ideally (but this is not compulsory), there must be at least one situation where this new proposal performs better that any of the 3 previous proposals. To describe your proposal you must include (with the same level of detail asked in the previous part of the assessment):

- **a)** A **description in English** of the idea behind the algorithm, in the same way proposals 1-3 were first explained. Use images if that facilitates the explanation.
- **b)** A **pseudocode description** of the algorithm
- **c)** The description of the **worst and best cases** of your proposal in terms of the running time.
- **d)** An **expression for the running time, T(N),** of your proposal for the best and worst cases.
- **e)** The growth function of the running time, for worst and best cases.
- **f)** The **Theta notation** for the worst and best cases.
- **g)** A discussion on how your proposal is better (or maybe not) than any of the proposals 1-3. Again, this is a **reflective writing exercise**. Make sure you comment on how your proposal compares to each of the 3 proposals given in this document.

As with Part 1, you can check the correctness of your work by implementing Proposal 4 (your proposal for Part 2) in C++ (or any programming language you want to use).

3. DOCUMENTS

Your submission must contain a **single PDF file** addressing the points:

- a)-f) of Part 1
- a)-g) of Part 2

4. MARKING CRITERIA

This mid-term assessment is worth 40% of your final mark. This percentage is further divided as follows:

- Work on Proposal 1: 6%
- Work on Proposal 2: 6%
- Work on Proposal 3: 6%
- Reflective writing comparing proposals 1-3: 5%
- Work on proposal 4: 12%
- Reflective writing comparing proposal 4 vs. proposals 1-3: 5%

For proposals 1-3, we will mark your work according to the set of criteria shown below.

CRITERION	**Not addressed (0-29%)**	**Attempted, with major omissions (30-69%)**	**Attempted, with minor omissions (70-99%)**	**Complete and correct work (100%)**	**Weight (out of 6%)**
Pseudocode					1.5%
Worst and best cases description					1%
T(N) (worst & best cases)					1%
Growth function					1%
Theta Notation (for worst & best cases)					1.5%

The reflective writing comparing proposals 1-3 will be marked as follows:

CRITERION	**Not addressed (0-29%)**	**Attempted, with major omissions (30-69%)**	**Attempted, with minor omissions (70-99%)**	**Complete work (100%)**	**Weight (out of 5%)**
The time complexity of the different proposals and what would be the best choice based exclusively on this					1%
Comparative summary of the 3 proposals in terms of time complexity					1.5%

Comparative study of the 3 proposals in terms of performance (number of requests accepted/rejected)					1%
Advanced discussion (e.g. performance discussion based on actual graphs obtained after implementing the system; or discussion how other algorithms (allocation, release) affect this evaluation)					2.5%

Proposal 4, which requires creativity from your side to devise a new algorithm, will be marked according to the following criteria.

CRITERION	**Not addressed (0-29%)**	**Attempted, with major omissions (30-69%)**	**Attempted, with minor omissions (70-99%)**	**Complete work (100%)**	**Weight (out of 12%)**
Clear English description					6%
Pseudocode					1.5%
Worst and best cases description					1%
T(N) (worst & best cases)					1%
Growth function					1%
Theta Notation (for worst & best cases)					1.5%

The reflective writing comparing proposal 4 to previous ones will be marked as follows:

CRITERION	**Not addressed (0-29%)**	**Attempted, with major omissions (30-69%)**	**Attempted, with minor omissions (70-99%)**	**Complete work (100%)**	**Weight (out of 5%)**
Comparative description of Proposal 4 with respect to Proposal 1					1%
Comparative description of Proposal 4 with respect to Proposal 2					1%
Comparative description of Proposal 4 with respect to Proposal 3					1%

Overall evaluation of the advantages/disadvantages of Proposal 4					2%

Notice that, if you do not submit a PDF file, you risk your work not being marked.