1.Given memory partitions of 100k,500k,200k,300k and 600k (in order), how  
would each of the First-fit, Best fit and worst –fit algorithms place processes of 212k,417k,112k and 426 k(in order)? which algorithm makes the most efficient use of the memory?

Memory allocation algorithms are methods of allocating memory to processes or threads that request it. There are different types of memory allocation algorithms, such as first-fit, best-fit, and worst-fit. These algorithms differ in how they search for a free memory partition that can satisfy a memory request¹²³.

## First-Fit Algorithm

The first-fit algorithm searches for the first free memory partition that is large enough to accommodate the memory request. If the partition is larger than the request, it is split into two parts: one part is allocated to the request and the other part remains free. If no suitable partition is found, the request is denied.

## Best-Fit Algorithm

The best-fit algorithm searches for the smallest free memory partition that is large enough to accommodate the memory request. If the partition is larger than the request, it is split into two parts: one part is allocated to the request and the other part remains free. If there are multiple partitions that can satisfy the request, the one with the smallest size is chosen. If no suitable partition is found, the request is denied.

## Worst-Fit Algorithm

The worst-fit algorithm searches for the largest free memory partition that is large enough to accommodate the memory request. If the partition is larger than the request, it is split into two parts: one part is allocated to the request and the other part remains free. If there are multiple partitions that can satisfy the request, the one with the largest size is chosen. If no suitable partition is found, the request is denied.

## Example

Let us consider an example of how each of these algorithms would place processes of 212k, 417k, 112k and 426k (in order) in memory partitions of 100k, 500k, 200k, 300k and 600k (in order). The following table shows how each algorithm would allocate memory to each process:

| Process | First-Fit | Best-Fit | Worst-Fit |

|---------|-----------|----------|-----------|

| 212k | 500k (288k left) | 300k (88k left) | 600k (388k left) |

| 417k | 600k (183k left) | 500k (83k left) | 500k (83k left) |

| 112k | 200k (88k left) | 100k (no split) | 300k (188k left) |

| 426k | Denied (no fit) | Denied (no fit) | Denied (no fit) |

As we can see from the table, none of the algorithms can allocate memory to all four processes. The first-fit algorithm fails to allocate memory to the last process because it cannot find a free partition that is large enough. The best-fit and worst-fit algorithms fail for the same reason.

## Efficiency

The efficiency of a memory allocation algorithm depends on several factors, such as:

- The size and number of memory requests and partitions

- The order and frequency of memory requests and releases

- The amount of internal and external fragmentation

- The time and space complexity of searching for a free partition

In general, there is no single algorithm that can guarantee optimal performance for all scenarios. However, some general observations can be made about each algorithm:

- The first-fit algorithm is fast and simple to implement, as it only requires a linear search for a free partition. However, it may suffer from high external fragmentation, as it tends to leave small gaps at the beginning of the memory.

- The best-fit algorithm may reduce external fragmentation by choosing the smallest fitting partition. However, it may also create many small holes that cannot be used by future requests. It also requires more time and space to search for and maintain a sorted list of free partitions.

- The worst-fit algorithm may reduce external fragmentation by choosing the largest fitting partition. However, it may also create large holes that are wasted if future requests are smaller than them. It also requires more time and space to search for and maintain a sorted list of free partitions.

2.Discuss in detail about

(i) Paging hardware - logical address to physical address mapping. (diagram)  
(ii) TLB

(iii) External fragmentation with solution

(i) Paging hardware - logical address to physical address mapping

Paging is a memory management technique that divides the physical memory into fixed-size units called frames and the logical memory into units of the same size called pages. [Paging allows the processes to use non-contiguous physical memory by mapping each page to a frame1](https://www.cse.iitb.ac.in/~mythili/os/anno_slides/lecture9.pdf)[2](https://www.geeksforgeeks.org/translation-lookaside-buffer-tlb-in-paging/)[3](https://lass.cs.umass.edu/~shenoy/courses/fall14/lectures/Lec13-part2.pdf).

The paging hardware is responsible for translating the logical addresses generated by the CPU into physical addresses that can access the main memory. The paging hardware consists of two main components: the page table and the translation lookaside buffer (TLB).

The page table is a data structure that stores the mapping between each page and its corresponding frame. The page table is maintained by the operating system and resides in the main memory. Each process has its own page table, which is loaded into a special register when the process is running.

The TLB is a cache that stores the most recently used page table entries. The TLB is part of the memory management unit (MMU) and resides in the CPU. The TLB reduces the overhead of accessing the page table in the main memory.

The following diagram shows how the paging hardware works:

The steps involved in logical address to physical address mapping are:

* The CPU generates a logical address, which consists of a page number and an offset within the page.
* The MMU checks if the page number is present in the TLB. If yes, this is a TLB hit and the MMU obtains the frame number from the TLB entry.
* If no, this is a TLB miss and the MMU accesses the page table in the main memory using the page number as an index. The MMU obtains the frame number from the page table entry and updates the TLB with this entry.
* The MMU combines the frame number and the offset to form the physical address, which can access the main memory.

(ii) TLB

TLB stands for translation lookaside buffer, which is a cache that stores the most recently used page table entries. The TLB is part of the memory management unit (MMU) and resides in the CPU. The TLB reduces the overhead of accessing the page table in the main memory[1](https://www.cse.iitb.ac.in/~mythili/os/anno_slides/lecture9.pdf)[2](https://www.geeksforgeeks.org/translation-lookaside-buffer-tlb-in-paging/)[3](https://lass.cs.umass.edu/~shenoy/courses/fall14/lectures/Lec13-part2.pdf).

The TLB works as follows:

* When a logical address is generated by the CPU, it consists of a page number and an offset within the page.
* The MMU checks if the page number is present in the TLB. If yes, this is a TLB hit and the MMU obtains the frame number from the TLB entry.
* If no, this is a TLB miss and the MMU accesses the page table in the main memory using the page number as an index. The MMU obtains the frame number from the page table entry and updates the TLB with this entry.
* The MMU combines the frame number and the offset to form the physical address, which can access the main memory.

The TLB has several advantages, such as:

* It improves performance by reducing memory access time for logical address translation.
* It reduces traffic on the memory bus by avoiding unnecessary accesses to main memory.
* It supports protection and sharing by storing additional information in each TLB entry, such as valid/invalid bit, protection bits, dirty bit, accessed bit, etc.

The TLB also has some challenges, such as:

* It has limited size and capacity, so it may not be able to store all possible page table entries for a process or multiple processes.
* It needs to be updated or invalidated when there are changes in page table entries due to allocation, deallocation, swapping, or context switching.
* It may cause inconsistency or coherence problems if there are multiple CPUs or caches that access or modify shared pages.

(iii) External fragmentation with solution

External fragmentation is a problem that occurs when there are many small gaps or holes in memory that are not usable by any process. External fragmentation happens when memory is allocated and deallocated dynamically using methods such as first-fit, best-fit, or worst-fit[1](https://www.cse.iitb.ac.in/~mythili/os/anno_slides/lecture9.pdf)[2](https://www.geeksforgeeks.org/translation-lookaside-buffer-tlb-in-paging/)[3](https://lass.cs.umass.edu/~shenoy/courses/fall14/lectures/Lec13-part2.pdf).

External fragmentation causes several issues, such as:

* It reduces memory utilization by wasting space that could be used by other processes.
* It increases memory access time by requiring more searches for free partitions that can satisfy memory requests.
* It may prevent processes from running if there is not enough contiguous memory available for their needs.

One possible solution for external fragmentation is compaction, which is a process of moving the allocated memory segments to one end of the memory and leaving a large free partition at the other end. Compaction eliminates external fragmentation by creating a single large hole that can be used by any process.

Compaction has some benefits, such as:

* It improves memory utilization by reclaiming the wasted space and making it available for allocation.
* It simplifies memory allocation by reducing the number of free partitions to search for.
* It enables processes to run if there is enough total memory available for their needs.

Compaction also has some drawbacks, such as:

* It is costly and time-consuming, as it requires moving many memory segments and updating their addresses.
* It may not be possible or feasible if the memory segments are not relocatable or if there are too many of them.
* It does not prevent external fragmentation from happening again, as it only solves the problem temporarily.

Another possible solution for external fragmentation is paging, which is a memory management technique that divides the physical memory into fixed-size units called frames and the logical memory into units of the same size called pages. Paging allows the processes to use non-contiguous physical memory by mapping each page to a frame[1](https://www.cse.iitb.ac.in/~mythili/os/anno_slides/lecture9.pdf)[2](https://www.geeksforgeeks.org/translation-lookaside-buffer-tlb-in-paging/)[3](https://lass.cs.umass.edu/~shenoy/courses/fall14/lectures/Lec13-part2.pdf).

Paging has some advantages, such as:

* It eliminates external fragmentation by using fixed-size memory units that can fit any process.
* It reduces internal fragmentation by choosing a small page size that minimizes the unused space within each page.
* It supports protection and sharing by storing additional information in each page table entry, such as valid/invalid bit, protection bits, dirty bit, accessed bit, etc.

Paging also has some challenges, such as:

* It adds overhead to memory access by requiring logical address to physical address translation using page table and TLB.
* It may cause thrashing or performance degradation if there are too many page faults or TLB misses due to insufficient physical memory or poor page replacement policy.
* It may increase disk I/O and swapping activity if there are frequent transfers of pages between main memory and secondary storage.