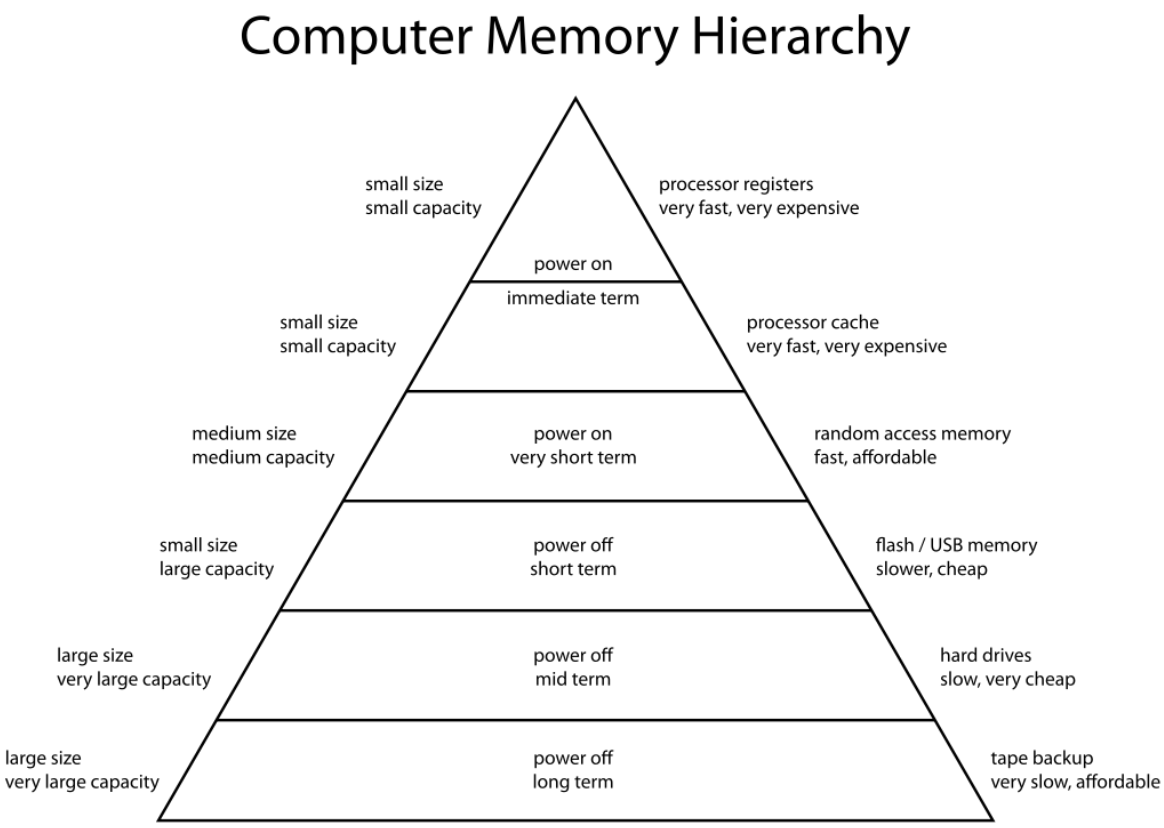
1. Memory hierarchy
    a. Shows trade-offs between quantity, speed, and cost
    b. Image below taken from Wikipedia



    c. Forms a pyramid
        i. At the top, costliest and smallest memories like registers and cache
            1. However, they are extremely quick to access
        ii. At the bottom, cheapest and largest memories like hard drives and tape
            1. Extremely slow to access, especially tape
    d. Key to success of the hierarchy
        i. As we move down the pyramid, we want to access the lower things less
        ii. Dependent on the idea of *locality*
    e. Locality types
        i. Temporal locality – if we accessed something previously, we're probably going to access it again soon
            1. Imagine an array
            2. We're probably going to go through it multiple times
        ii. Spatial locality – if we access something, odds are we're going to access something near it
            1. Imagine an array again
            2. If we touch the first element, we're probably going to access the second and third ones as well
    f. Locality is the key behind why caches work so well

2. Cache principles
    a. Cache holds a much smaller subset of RAM's contents
        i. Bring in values from RAM while executing a program
        ii. If we need a value, look in cache first before going to memory
        iii. Benefit from having cache because it is faster than RAM, and closer to the CPU
        iv. However, cache is smaller than RAM, and costs more per bit than RAM
    b. Terminology
        i. Memory *block* – unit of main memory stored in a cache line
        ii. Cache *line* – basic unit of a cache
            1. Contains a block of memory
            2. Also contains a tag, and control bits to determine when line should be replaced
        iii. Cache *tag* – number stored with a line
            1. Along with the line's position in the cache, determines the address of the block from main memory
        iv. Cache *hit* – data that the CPU asks for is in the cache
        v. Cache *miss* – data that the CPU asks for is *not* in the cache
        vi. *Valid* bit – data currently inside a cache line is legitimate
            1. When we first boot up the cache it contains garbage values
            2. Without a valid bit cache could say there's a hit when there really wasn't one
        vii. *Eviction* – removing memory from the cache and replacing with new memory
        viii. *Dirty* bit – line contains updated data that differs from the corresponding main memory
            1. 0 when first pulling a line from memory
            2. If we write to that cache line to make changes, set dirty bit to 1
            3. Indicates that before evicting the line from cache, must copy back to main memory
                a. Otherwise, would lose changes to that data

3. Cache addressing
    a. What type of addresses does the cache use?
    b. Cache addresses can be *physical*, using actual memory addresses
        i. Slower, as the cache must wait for the MMU
        ii. Memory management unit (MMU) converts virtual to physical addresses
        iii. However, physically addressed caches don't have to deal with the aliasing problem below
    c. Can also be *logical*, using the same virtual addresses that the process uses
        i. Faster, as the cache doesn't need to wait for MMU
            1. Can reuse the same address that the process was using
        ii. However, these caches encounter the *aliasing* problem
            1. Same virtual address can be used in multiple user processes
            2. Cache would say there's a hit
            3. However, that data doesn't correspond to process currently running
            4. Need to flush cache every time we switch processes
                a. Also known as a *context switch*, which you will hear constantly in ECS 150
            5. Incur a performance penalty every time we flush cache
                a. Lose all our built-up cache data
                b. Must go back to RAM to fetch new values