1. Dynamic Memory Allocation
   1. Reasons to use
      1. Lots of memory is needed.
         1. Insufficient memory from stack
      2. For the memory allocated to still be allocated when the function within which the memory is allocated finishes.
   2. Example
      1. int n; // however many integers are needed
      2. {
         1. scanf(“%d”, &n);
         2. int \* values = malloc(n \* sizeof(int)); //malloc returns address of first block of memory available
            1. Indexed from 0 to n-1
            2. values works like a regular array
      3. }
      4. …
      5. free(values); // **Every single malloc() has an equal and opposite free()**
2. Searching for a value in a *sorted* array
   1. Example
      1. int search(int \* array, int length, int search\_val)
      2. {
      3. int i;
      4. for (i = 0; i < length; i++)
         1. if (array[i] == search\_val)
         2. return 1;
   2. Runtime
      1. 0(*n*)
      2. Not very efficient
3. Binary search on a *sorted array*
   1. Logic
      1. Keep a low and high index of the array
      2. For each guess, guess halfway way in between 2 indexes
   2. Example
      1. int search(int \* array, int length, int search\_val)
      2. {
         1. int low = 0, mid, high = LENGTH – 1;
         2. while (low <= high)
         3. {
            1. mid = (low + high)/2;
            2. if(search\_val > array[mid])

low = mid + 1;

* + - * 1. else if (search\_val < array[mid])

high = mid – 1;

* + - * 1. else

return 1; //value is found

* + - 1. }
      2. return 0;
    1. }
  1. Runtime
     1. Array after each comparison
        1. *n*
        2. *n*/2
        3. *n*/4…
        4. *n*/2*k*, where *k* is the number of times the while loop runs
     2. n/2k = 1 🡪 *k* = log2*n*
        1. *k* is the worst possible number of times that the program can run
     3. 0(lg*n*)