CS302 Lecture notes
Binary Search

- James S. Plank

- Directory: **/home/plank/cs302/Notes/BinarySearch**
- August 23, 2023
- Latest revision: August 24, 2023
- Git Hash: 3e3a04815c02a4bc1523ea82e75e0604202cf082

Reference Material Online / Topcoder / Leetcode Problem(s)

- Binary Search from Wikipedia
- Leetcode.com problem 540: Single Element in a Sorted Array. Detailed writeup, but no solution given.
- Leetcode.com problem 2300: Successful Pairs of Spells and Potions. Detailed writeup, but no solution given.
- Topcoder SRM 721, D1, 250-Pointer (RememberWords). Brief writeup, but no solution given.
- Leetcode.com problem 4: "Median-Of-Two-Sorted-Arrays". Detailed writeup, but no solution given. This is a tough one!
- Leetcode.com problem 1011. Capacity To Ship Packages Within D Days. No writeup or solution -- this one is similar to "Koko Eating Bananas" below, plus the Leetcode editorial is unlocked by default, so that can help if you need it.
- Leetcode.com problem 74. Search a 2D Matrix. Two binary searhes. No writeup or solution.
- Leetcode.com problem 1802. Maximum Value at a Given Index in a Bounded Array. No writeup, but the Leetcode editorial is unlocked by default, so that can help if you need it.

What do you need to know for the exam?

- How to perform a binary search.
- Its running time.
- How to identify problems as being solvable with binary search.

Overview

Binary search is a very powerful problem-solving technique. The idea is simple. Given a collection of n items and problem to solve, you solve the problem by throwing away half of the items, and then solving the problem on the half that you have remaining.

I know that's a bit vague, but that's the general principle. The running time is straightforward, too: At each step of your problem, you're throwing away half of the items, so after $log(n)$ steps, you'll be left with one item, and you're done.

That means your binary search is generally $O(log\ n)$, which is smoking fast. I say "generally", because sometimes you need to do extra stuff during a step, and that can increase your overhead.

Conceptually Easy, But Typically Harder

When you go to implement a binary search, you'll find that your code is often buggy, and you need some inelegant edge conditions. That's normal, and you should be ready for it. The reason that I decided to teach binary search in CS302 (starting 2023) is that every time I had to do a binary search problem in Leetcode, it took extra attention to detail and extra testing. I have some helpful advice, I believe, in these lecture notes, but experience is always the best teacher.

A Canoncial Example

Here's a canonical example. The file txt/words.txt contains a dictionary of 24,853 words, all lower-case, sorted:

Copyright © 2023, James S. Plank. All rights reserved.

```
UNIX> wc txt/words.txt
24853   24853  205393 txt/words.txt                 # There are 24,853 words
UNIX> head -n 5 txt/words.txt                       # It looks sorted from the beginning
aaa
aaas
aarhus
aaron
aau
UNIX> tail -n 5 txt/words.txt                       # And end
zoroastrian
zounds
zucchini
zurich
zygote
UNIX> sort txt/words.txt > tmp.txt                  # We demonstrate that it is indeed sorted.
UNIX> diff txt/words.txt tmp.txt
UNIX>
```

Let's write a program, in src/dict_bsearch.cpp. It's a straightforward binary search -- please read the comments inline for explanation.

```
/* src/dict_bsearch.cpp.  This reads a dictionary of words from a file into a vector of
   strings.
It sorts the vector, if not already sorted, and then it reads words from standard input
It uses binary search to determine whether each of these words is in the dictionary. */

#include <vector>
#include <algorithm>
#include <iostream>
#include <fstream>
using namespace std;

class Bsearch {
public:
    void Create(const string &filename);        // Create the vector from the file.
    bool Find(const string &word) const;        // Return whether a word is in the vector.
protected:
    vector <string> words;
};

/* Create() is straightforward -- it reads each words into a vector, and while doing so,
determines whether the vector is sorted.  If it is not, then it is sorted at the
end of Create(). */

void Bsearch::Create(const string &filename)
{
ifstream fin;
bool sorted;
string w;

sorted = true;

fin.open(filename.c_str());
if (fin.fail()) throw (string) "Could not open " + filename;
```


Copyright © 2023, James S. Plank. All rights reserved.

```
while (fin >> w) {
    if (words.size() > 0 && w < words[words.size()-1]) sorted = false;
    words.push_back(w);
}
if (!sorted) sort(words.begin(), words.end());
}

/* Here's the binary search.  It keeps track of three variables:

l = the index of the smallest word that we are considering.
h = the index of the largest word that we are considering.
m = the middle of l and h

It iterates by looking at words[m], and using that value to either return,
discard the lower half of elements or discard the higher half of elements.
*/

bool Bsearch::Find(const string &word) const
{
int l, h, m;

if (words.size() == 0) return false;

l = 0;                          // Initially, we consider the entire vector
h = words.size() - 1;

while (l <= h) {
    m = (l + h) / 2;
    // printf("l:%d(%s)  m:%d(%s)  h:%d(%s)\n",
    //          l, words[l].c_str(), m, words[m].c_str(), h, words[h].c_str());
    if (words[m] == word) return true;
    if (words[m] > word) h = m-1;      // Throw away the top half
    if (words[m] < word) l = m+1;      // Throw away the bottom half
}
return false;

}

/* The main() is straightfoward -- create the dictionary from the file, then find each
word on standard input. */

/* I'm not including it here in the lecture notes. */
```

Let's test by uncommenting the **printf()** statement inside **Find()**, and doing some small examples:

Copyright © 2023, James S. Plank. All rights reserved.

```
UNIX> wc txt/words-12.txt          # My dictionary has 12 words
    12      12      99 txt/words-12.txt
UNIX> cat txt/words-12.txt
attention
debtor
efficient
goldenseal
highwaymen
hogan
moth
rebutted
salsify
stud
wakeful
woodpeck

         # I'm going to find attention, debtor, efficient and hogan, which are all there:

UNIX> echo attention | bin/dict_bsearch txt/words-12.txt y
l:0(attention)  m:5(hogan)  h:11(woodpeck)
l:0(attention)  m:2(efficient)  h:4(highwaymen)
l:0(attention)  m:0(attention)  h:1(debtor)
attention: found
Found: 1 of 1

UNIX> echo debtor | bin/dict_bsearch txt/words-12.txt y
l:0(attention)  m:5(hogan)  h:11(woodpeck)
l:0(attention)  m:2(efficient)  h:4(highwaymen)
l:0(attention)  m:0(attention)  h:1(debtor)
l:1(debtor)  m:1(debtor)  h:1(debtor)
debtor: found
Found: 1 of 1

UNIX> echo efficient | bin/dict_bsearch txt/words-12.txt y
l:0(attention)  m:5(hogan)  h:11(woodpeck)
l:0(attention)  m:2(efficient)  h:4(highwaymen)
efficient: found
Found: 1 of 1

UNIX> echo hogan | bin/dict_bsearch txt/words-12.txt y
l:0(attention)  m:5(hogan)  h:11(woodpeck)
hogan: found
Found: 1 of 1

         # Now aaa, zzz and mmm, which are not there:

UNIX> echo aaa | bin/dict_bsearch txt/words-12.txt y
l:0(attention)  m:5(hogan)  h:11(woodpeck)
l:0(attention)  m:2(efficient)  h:4(highwaymen)
l:0(attention)  m:0(attention)  h:1(debtor)
aaa: not-found
Found: 0 of 1
```


Copyright © 2023, James S. Plank. All rights reserved.

```
UNIX> echo zzz | bin/dict_bsearch txt/words-12.txt y
l:0(attention)  m:5(hogan)  h:11(woodpeck)
l:6(moth)  m:8(salsify)  h:11(woodpeck)
l:9(stud)  m:10(wakeful)  h:11(woodpeck)
l:11(woodpeck)  m:11(woodpeck)  h:11(woodpeck)
zzz: not-found
Found: 0 of 1

UNIX> echo mmm | bin/dict_bsearch txt/words-12.txt y
l:0(attention)  m:5(hogan)  h:11(woodpeck)
l:6(moth)  m:8(salsify)  h:11(woodpeck)
l:6(moth)  m:6(moth)  h:7(rebutted)
mmm: not-found
Found: 0 of 1

UNIX>
```

It's a good idea to go over the examples above and look at the indices, to see how it hones the search space at each step. Let's look at a bigger example to see what happens when it tries to find "jjj" in **txt/words.txt**. I'm going to have that print statement print $(h - l)$ at each step, so you can see how it roughly halves at each step:

```
UNIX> echo jjj | bin/dict_bsearch txt/words.txt y
h-l:24852 l:0(aaa)  m:12426(jewelry)  h:24852(zygote)
h-l:12425 l:12427(jewett)  m:18639(refractory)  h:24852(zygote)
h-l:6211 l:12427(jewett)  m:15532(nightfall)  h:18638(refractometer)
h-l:3104 l:12427(jewett)  m:13979(mambo)  h:15531(nightdress)
h-l:1551 l:12427(jewett)  m:13202(legendary)  h:13978(maltreat)
h-l:774 l:12427(jewett)  m:12814(knapsack)  h:13201(legend)
h-l:386 l:12427(jewett)  m:12620(kamchatka)  h:12813(knapp)
h-l:192 l:12427(jewett)  m:12523(joyous)  h:12619(kalmuk)
h-l:95 l:12427(jewett)  m:12474(johns)  h:12522(joyful)
h-l:46 l:12427(jewett)  m:12450(joanna)  h:12473(johnny)
h-l:22 l:12427(jewett)  m:12438(jimenez)  h:12449(joan)
h-l:10 l:12439(jimmie)  m:12444(jitterbug)  h:12449(joan)
h-l:4 l:12445(jitterbugger)  m:12447(jittery)  h:12449(joan)
h-l:1 l:12448(jive)  m:12448(jive)  h:12449(joan)
h-l:0 l:12449(joan)  m:12449(joan)  h:12449(joan)
jjj: not-found
Found: 0 of 1
UNIX>
```

How does binary search compare with sets and unordered sets?

If I asked you to write the program above without binary search, but instead using the standard template library, I hope you would consider the following:

- A **set**. Creating the set is $O(n\ log\ n)$, where n is the number of words in the dictionary, and then performing the **Finds** is $O(m\ log\ n)$, where m is the number of **Finds**.
- An **unordered_set**. This employs a hash table under the hood -- creating it is $O(n)$ and performing the **Finds** is $O(m)$.

That argues for the **unordered_set**. Where does binary search fit it? Well, creating the vector is $O(n)$ if it's sorted, and $O(n\ log\ n)$ if it's not. And performing the **Finds** is $O(m\ log\ n)$. In other words, identical to the **set**.

Copyright © 2023, James S. Plank. All rights reserved.

I have implemented the **set** and **unordered_set** code in src/dict_set.cpp and src/dict_uset.cpp respectively. To test, I have created txt/test.txt, which has 12,000 words from txt/words.txt, and 12,000 words that are not in txt/words.txt. Let's see how they compare:

```
UNIX> make clean
rm -f bin/*
UNIX> make bin/dict_bsearch bin/dict_set bin/dict_uset
g++ -o bin/dict_bsearch -Wall -Wextra -std=c++11 src/dict_bsearch.cpp
g++ -o bin/dict_set -Wall -Wextra -std=c++11 src/dict_set.cpp
g++ -o bin/dict_uset -Wall -Wextra -std=c++11 src/dict_uset.cpp
UNIX> time bin/dict_bsearch txt/words.txt < txt/test.txt n
Found: 12000 of 24000

real       0m0.151s
user       0m0.147s
sys        0m0.003s

UNIX> time bin/dict_set txt/words.txt < txt/test.txt n
Found: 12000 of 24000

real       0m0.226s
user       0m0.220s
sys        0m0.004s

UNIX> time bin/dict_uset txt/words.txt < txt/test.txt n
Found: 12000 of 24000
real       0m0.089s
user       0m0.084s
sys        0m0.003s

UNIX>
```

Predictably, the **unordered_set** was the fastest. The binary search is significantly faster than the **set**, even though they have the same big-O. Part of that is because we don't sort the words (they are already sorted), but the significant savings actually come from memory. The **set** uses a tree data structure, which has a lot of pointers and extra memory. The binary search simply uses the vector.

Keep that in mind.

Using start/size rather than low/high

Rather than using low/high (or left-right) indices, you can specify the part of the vector under review using:

- **Start:** This is the index of the first element of the vector that we are testing.
- **Size:** This is the size of the region of the vector that we are testing.

After spending quite a bit of time implementing binary searches using both types of information, I have concluded that I like keeping track of **start** and **size** better than **low** and **high**. The main reason is that so long as you make sure that **size** is greater than zero, then **words**[**start**] is always valid. I find that comforting. Here's the implementation of **Find()** using **start/size**. The code is in src/dict_bs_ss.cpp:

Copyright © 2023, James S. Plank. All rights reserved.

```
bool Bsearch::Find(const string &word) const
{
    int start, size, mid;

    start = 0;
    size = words.size();
    if (size == 0) return false;

    while (size > 1) {
    mid = start + size/2;

    /* I guess I like how this code translates logically: */

    if (words[mid] > word) {     /* If word is not in the second half... */
        size /= 2;               /* Discard the second half. */
    } else {
        start += size/2;         /* Otherwise discard the first half. */
        size -= size/2;          /* Note this handles even and odd sizes correctly. */
    }
    }
    return (words[start] == word);
}
```

It's faster than the previous code (it was about 0.151 rather than 0.114 here):

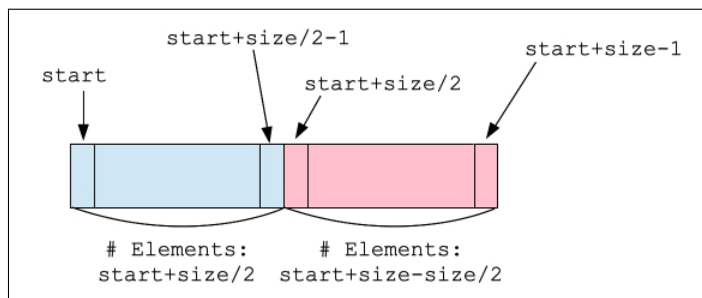
```
UNIX> time bin/dict_bs_ss txt/words.txt < txt/test.txt n
Found: 12000 of 24000

real       0m0.114s.
user       0m0.110s.
sys        0m0.003s
UNIX>
```

Plus it has another advantage of the previous code -- if the vector contains duplicates, this will always return the last of the duplicates.

Taking a look at that visually

I find this graphic really helpful when thinking about binary searches:



Your goal is to either eliminate the light blue region or the pink region. Sometimes that's pretty natural, as in the code above. Sometimes less so. The next section is an example.

Copyright © 2023, James S. Plank. All rights reserved.

Less obvious uses of binary search -- optimization

When you're trying to spot a binary search solution to a problem, see if you can frame the problem in the following way:

Let me define a function $f(v)$ whose answer is "yes/no". Let's suppose that $f(v)$ is $O(n)$. Moreover, suppose that there is a value v_{opt}, such that for all $v < v_{opt}, f(v)$ is "yes", and for all $v > v_{opt}, f(v)$ is "no". Then we can use binary search on v to fine v_{opt}. The running time is going to be $O(n\ log\ v)$.

A good example of this is Leetcode Medium problem #875: "Koko Eating Bananas". Here's the problem on Leetcode if you want to try it yourself: `https://leetcode.com/problems/koko-eating-bananas/`.

Here's a summary of the problem:

- You are given a vector of integers called **piles**. Its values are between 1 and 10^9, and its length is between 1 and $10,000$.
- You are given a value **h**, which is between **piles.size()** and 10^9.
- You are to derive a value **k**, which works as follows.
- At each timestep, you may reduce the value of a pile by **k**. You may not reduce the value below zero, and you may not reduce the value of more than one pile in a timestep.
- What is the minimum value of **k** that allows you to reduce all of the piles to zero in at most **h** timesteps.

Let's work through the Leetcode examples:

- Example 1: **piles = [3, 6, 7, 11], h=8**.

  The answer is 4:

  - 1 timestep for **piles[0]** = 3.
  - 2 timesteps for **piles[1]** = 6.
  - 2 timesteps for **piles[2]** = 7.
  - 3 timesteps for **piles[3]** = 11.

  That's 8 timesteps. You'll note that if you set **k** to 3, then it will take you 10 timesteps.

- Example 2: **piles = [30, 11, 23, 4, 20], h=5**.

  The answer is 30, because you have to reduce each pile to zero in a single timestep.

- Example 3: **piles = [30, 11, 23, 4, 20], h=6**.

  The answer is 23. That way, you get piles 1 through 4 in one timestep, and pile 0 in two tiemsteps.

This is a typical topcoder/leetcode problem, because your first inclination is to think in terms how you break up the individual piles. However, resist that temptation. Instead, focus on the observations which point toward binary search:

1. If you set k to zero, you'll never succeed.
2. If you set k to the maximum pile size, you'll finish in exactly **piles.size()** timesteps. In other words, youll always succeed.
3. If you have a value of k and you increase it, then the number of timesteps will stay the same or be reduced.
4. If you have a value of k and you decreases it, then the number of timesteps will stay the same or be increased.

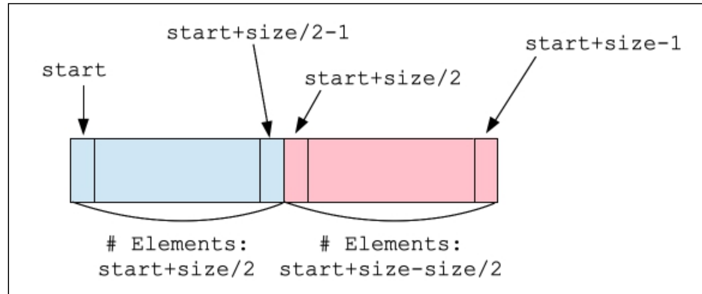
This gives you all of the conditions that you need for binary search. Let $f(k)$ = "yes" if you can succeed with a value of k, and "no" if you cannot. You know that k_{opt}. is going to be greater than zero and less than or equal to the maximum pile size. So use binary search to find k_{opt}.

Copyright © 2023, James S. Plank. All rights reserved.

Here are the broad strokes:

- Test the middle element for k. If you can succeed, discard the higher half. If you can't, discard the lower half.
- An important thing is that you *know* that your answer is in the region that you're considering. Thus, the binary search works.

It's actually a little more complex than that. Let's return to the picture above:



Suppose we set **mid** to **start+size/2** and then we test to see **mid** is a successful value of k. If it is, then we can throw out all of the values greater than **mid**. That's not what we want. Instead, set **mid** to **start+size/2-1** -- that's the last element in the blue region above. If it is successful, then we can throw out the pink region. If it's not, then we can throw out the blue region. That's just what we want!

If you want to work on this yourself, you can start with the skeleton program in src/Koko_Skeleton.cpp. This sets up a driver program and a skeleton (incorrect) implementation of the Leetcode method "minEatingSpeed". If we compile and run, it always returns zero:

```
UNIX> g++ src/Koko_Skeleton.cpp
UNIX> echo 3 6 7 11    8 | ./a.out        # Example 1
0
UNIX> echo 30 11 23 4 20   5 | ./a.out    # Example 2
0
UNIX>
```

Below is the commented answer, using binary search. It's in src/Koko_Solution.cpp. Here's the minEatingSpeed method:

```
int Solution::minEatingSpeed(vector<int>& piles, int h)
{
    int start, size, maxpile, mid;
    int i;
    long long timesteps, for_pile;

    /* Calculate the maximum pile size. */

    maxpile = piles[0];
    for (i = 0; i < piles.size(); i++) if (piles[i] > maxpile) maxpile = piles[i];

    /* We want our range to start at one and end at maxpile (including maxpile).
       So we set start to 1 and size to maxpile. Remember size means that start+size
       is one past the last element in our region. */

    start = 1;
```


Copyright © 2023, James S. Plank. All rights reserved.

```
    while (size > 1) {

    /* You want to test the highest value in the first half of the values (
        the last value in the blue region of the picture. */

    mid = start + size/2 - 1;

    /* Timesteps is the total timesteps if k is set to mid. */

    timesteps = 0;
    for (i = 0; i < piles.size(); i++) {
        for_pile = piles[i] / mid;
        if (piles[i]%mid != 0) for_pile++;
        timesteps += for_pile;
    }

    // printf("Start: %d.  Size: %d  Mid: %d.  Timesteps: %lld\n", start, size, mid,
        timesteps);

    /* If timesteps is too big, then you know that the answer
        is in the second half of the range.  You can throw out the
        first half. */

    if (timesteps > h) {
        start += size/2;
        size -= size/2;

    /* Otherwise, the answer is in the first half of the range, so toss out the second
        half. */

    } else {
        size = size/2;
    }
    }
    return start;
}
```

(If you care, that solution was pretty much smack at 50% in terms of speed on Leetcode).

Another Optimization Problem

This is Leetcode problem 2616: "Minimize the Maximum Difference of Pairs." Here's the link: https://leetcode.com/problems/minimize-the-maximum-difference-of-pairs/.

Here's my description:

- You are given a vector of integers between 0 and 10^9, whose size is between 1 and $100,000$.
- You are also given an integer p between 0 and half the vector's length.
- Your goal is to find p distinct pairs of numbers in the vector, whose maximum difference is minimized.
- The vector may contain duplicates -- if there are d duplicate values of the number x, then you can use x up to d times in your pairs.
- Return the minimum maximum difference. (ha ha).

Copyright © 2023, James S. Plank. All rights reserved.

Examples help. Let's use their example one:

```
nums = [10, 1, 2, 7, 1, 3 ]
p = 2
```

So we need to find two pairs and minimize the maximum difference within a pair. The answer here is 1 -- $\{(1,1),(2,3)\}$. The two differences are 0 and 1, so the maximum difference is 1. It's the best you can do, so that's the answer.

To formulate this as a binary search problem, let's define $f(v)$ as follows:

- $f(v)$ is "yes" if you can find p distinct pairs whose differences are all less than or equal to v.
- $f(v)$ is "no" if you cannot find p distinct pairs whose differences are all less than or equal to v.

It should be clear that there is a v_{opt} here. Moreover, we know that there is a value of f for which $f(v)$ is "yes" -- it's the maximum value of the vector.

So -- continuing with the problem formulation: If you can implement $f(v)$ in $O(n)$ time (where n is the size of the vector), then you can use binary search to find v_{opt}.

So we need to implement $f(v)$. To do that we can sort the vector, and the proceed greedily. Look at $nums[0]$. If $(nums[1] - nums[0]) \leq k$, then we count it as a pair and move onto $nums[2]$. Otherwise, we ignore $nums[0]$ and move onto $nums[1]$. You can prove to yourself that if there are p pairs, then this algorithm will find it. I won't do that formally, but you should give it some thought to convince yourself that this is true.

Let's code it up. The Leetcode class/method is:

```
class Solution {
  public:
    int minimizeMax(vector <int> &nums, int p);
};
```

In src/Min_The_Max_Skeleton.cpp I have a skeleton that reads in the array and p and then calls **minimizeMax()**. As always, it compiles and runs, but not correctly:

```
UNIX> make bin/Min_The_Max_Skeleton
g++ -o bin/Min_The_Max_Skeleton -Wall -Wextra -std=c++11 src/Min_The_Max_Skeleton.cpp
UNIX> echo 10 1 2 7 1 3  2 | bin/Min_The_Max_Skeleton
0
UNIX>
```

We'll program incrementally. First, we'll write $f(v)$, which returns whether you can find p pairs, each of whose difference is $\leq v$. It's in src/Min_The_Max_Write_F.cpp:

Copyright © 2023, James S. Plank. All rights reserved.

```
class Solution {
  public:
    int minimizeMax(vector <int> &nums, int p);
    bool f(int v, const vector <int> &nums, int p);
};

/* f(v, nums, p) returns true if there are p pairs in nums whose differences is less than
   or equal to p. */

bool Solution::f(int v, const vector <int> &nums, int p)
{
  size_t i;
  int np;

  np = 0;
  for (i = 0; i < nums.size() && np < p; i++) {
    if (i < nums.size()-1 && nums[i+1] - nums[i] <= v) {
      np++;
      i++;
    }
  }
  return (np >= p);
}

/* This just lets us test f(). */

int Solution::minimizeMax(vector <int> &nums, int p)
{
  int i;

  sort(nums.begin(), nums.end());
  for (i = 0; i < 10; i++) printf("%d %s\n", i, (f(i, nums, p)) ? "yes" : "no");
  return 0;
}
```

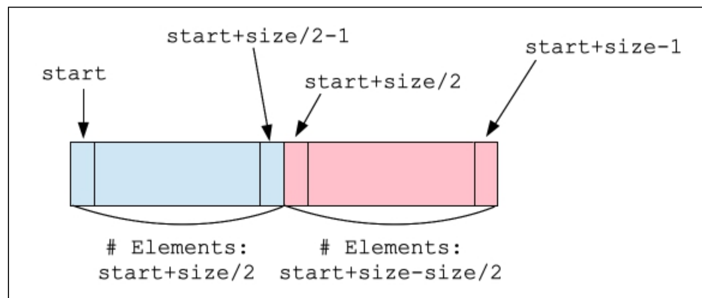
Let's test using *nums* from the first example.

```
UNIX> make bin/Min_The_Max_Write_F
g++ -o bin/Min_The_Max_Write_F -Wall -Wextra -std=c++11 src/Min_The_Max_Write_F.cpp
bash: bin/Min_The_Max_Skeleton: No such file or directory
UNIX>  echo 10 1 2 7 1 3  2 | bin/Min_The_Max_Write_F
0 no
1 yes     # This is the correct answer
2 yes
3 yes
4 yes
5 yes
6 yes
7 yes
8 yes
9 yes
0
```


Copyright © 2023, James S. Plank. All rights reserved.

```
UNIX>  echo 10 1 2 7 1 3  3 | bin/Min_The_Max_Write_F
0 no
1 no
2 no
3 yes    # As is this -- (10,7), (1,2), (1,3) -- among other solutions
4 yes
5 yes
6 yes
7 yes
8 yes
9 yes
0
UNIX>
```

Now we write the binary search. Let's consult the picture -- should we test the last entry of the blue section, or the first entry of the pink section?



Well, if we test the last entry of the blue section, then a "yes" answer lets us discard the pink section, and a "no" answer lets us discard the blue section. That's what we want! Here's the code: (src/Min_The_Max_Solution.cpp)

Copyright © 2023, James S. Plank. All rights reserved.

```
int Solution::minimizeMax(vector <int> &nums, int p)
{
    size_t i;

    int middle;
    int max;
    int start, size;

    sort(nums.begin(), nums.end());

    /* Calculate the maximum value in nums. */
    max = nums[0];
    for (i = 1; i < nums.size(); i++) if (nums[i] > max) max = nums[i];

    start = 0;
    size = max+1;    // Since 0 is a valid answer, and we want size to be one past the
                     // highest valid answer, we set size to max + 1.

    while (size > 1) {
    middle = start + size/2 - 1;
    if (f(middle, nums, p)) {
        size = size/2;
    } else {
        start += size/2;
        size -= size/2;
    }
    }
    return start;
}
```

Copyright (c) 2023, James S. Plank

Copyright © 2023, James S. Plank. All rights reserved.