Eric Roberts
CS 106A

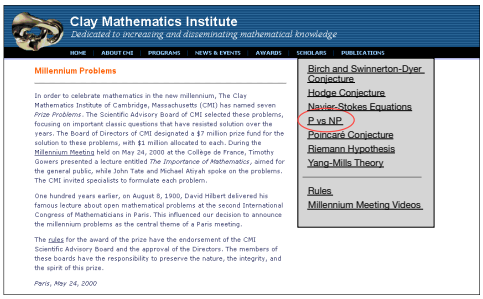
Handout #59
May 30, 2012

Searching and Sorting

Searching and Sorting

Eric Roberts
CS 106A
May 30, 2012

The Millennium Challenge Problems



Searching

- Chapter 12 looks at two operations on arrays—*searching* and *sorting*—both of which turn out to be important in a wide range of practical applications.
- The simpler of these two operations is ***searching***, which is the process of finding a particular element in an array or some other kind of sequence. Typically, a method that implements searching will return the index at which a particular element appears, or −1 if that element does not appear at all. The element you're searching for is called the ***key***.
- The goal of Chapter 12, however, is not simply to introduce searching and sorting but rather to use these operations to talk about algorithms and efficiency. Many different algorithms exist for both searching and sorting; choosing the right algorithm for a particular application can have a profound effect on how efficiently that application runs.

Linear Search

- The simplest strategy for searching is to start at the beginning of the array and look at each element in turn. This algorithm is called ***linear search***.
- Linear search is straightforward to implement, as illustrated in the following method that returns the first index at which the value `key` appears in `array`, or −1 if it does not appear at all:

```
private int linearSearch(int key, int[] array) {
   for (int i = 0; i < array.length; i++) {
      if (key == array[i]) return i;
   }
   return -1;
}
```

Simulating Linear Search

```
public void run() {
   private int linearSearch(int key, int[] array) {
      for ( int i = 0 ; i < array.length ; i++ ) {
         if (key == array[i]) return i;
      }
      return -1;
   }
```

i	key	array
10	27	

2	3	5	7	11	13	17	19	23	29
0	1	2	3	4	5	6	7	8	9

LinearSearch

```
linearSearch(17) -> 6
linearSearch(27) -> -1
```

Searching for Area Codes

- To illustrate the efficiency of linear search, it is useful to work with a somewhat larger example.
- The example on the next slide works with an array containing many of the area codes assigned to the United States.
- The specific task in this example is to search this list to find the area code for the Silicon Valley area, which is 650.
- The linear search algorithm needs to examine each element in the array to find the matching value. As the array gets larger, the number of steps required for linear search grows in the same proportion.
- As you watch the slow process of searching for 650 on the next slide, try to think of a more efficient way in which you might search this particular array for a given area code.

Linear Search (Area Code Example)

201	202	203	205	206	207	208	209	210	212	213	214	215	216	217	218	219	224	225	228	229	231
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
234	239	240	248	251	252	253	254	256	260	262	267	269	270	276	281	283	301	302	303	304	305
22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
307	308	309	310	312	313	314	315	316	317	318	319	320	321	323	325	330	331	334	336	337	339
44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65
347	351	352	360	361	364	385	386	401	402	404	405	406	407	408	409	410	412	413	414	415	416
66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87
417	419	423	424	425	430	432	434	435	440	443	445	469	470	475	478	479	480	484	501	502	503
88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109
504	505	507	508	509	510	512	513	515	516	517	518	520	530	540	541	551	559	561	562	563	564
110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131
567	570	571	573	574	575	580	585	586	601	602	603	605	606	607	608	609	610	612	614	615	616
132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153
617	618	619	620	623	626	630	631	636	641	646	650	651	660	661	662	678	682	701	702	703	704
154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
706	707	708	712	713	714	715	716	717	718	719	720	724	727	731	732	734	740	754	757	760	762
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197
763	765	769	770	772	773	774	775	779	781	785	786	801	802	803	804	805	806	808	810	812	813
198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219
814	815	816	817	818	828	830	831	832	835	843	845	847	848	850	856	857	858	859	860	862	863
220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241
864	865	870	878	901	903	904	906	907	908	909	910	912	913	914	915	916	917	918	919	920	925
242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263
928	931	936	937	940	941	947	949	951	952	954	956	959	970	971	972	973	978	979	980	985	989
264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285

The Idea of Binary Search

- The fact that the area code array is in ascending order makes it possible to find a particular value much more efficiently.
- The key insight is that you get more information by starting at the middle element than you do by starting at the beginning.
- When you look at the middle element in relation to the value you're searching for, there are three possibilities:
  - If the value you are searching for is greater than the middle element, you can discount every element in the first half of the array.
  - If the value you are searching for is less than the middle element, you can discount every element in the second half of the array.
  - If the value you are searching for is equal to the middle element, you can stop because you've found the value you're looking for.
- You can repeat this process on the elements that remain after each cycle. Because this algorithm proceeds by dividing the list in half each time, it is called ***binary search***.

Binary Search (Area Code Example)

Binary search needs to look at only eight elements to find 650.

201	202	203	205	206	207	208	209	210	212	213	214	215	216	217	218	219	224	225	228	229	231
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
234	239	240	248	251	252	253	254	256	260	262	267	269	270	276	281	283	301	302	303	304	305
22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
307	308	309	310	312	313	314	315	316	317	318	319	320	321	323	325	330	331	334	336	337	339
44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65
347	351	352	360	361	364	385	386	401	402	404	405	406	407	408	409	410	412	413	414	415	416
66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87
417	419	423	424	425	430	432	434	435	440	443	445	469	470	475	478	479	480	484	501	502	503
88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109
504	505	507	508	509	510	512	513	515	516	517	518	520	530	540	541	551	559	561	562	563	564
110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131
567	570	571	573	574	575	580	585	586	601	602	603	605	606	607	608	609	610	612	614	615	616
132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153
617	618	619	620	623	626	630	631	636	641	646	650	651	660	661	662	678	682	701	702	703	704
154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
706	707	708	712	713	714	715	716	717	718	719	720	724	727	731	732	734	740	754	757	760	762
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197
763	765	769	770	772	773	774	775	779	781	785	786	801	802	803	804	805	806	808	810	812	813
198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219
814	815	816	817	818	828	830	831	832	835	843	845	847	848	850	856	857	858	859	860	862	863
220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241
864	865	870	878	901	903	904	906	907	908	909	910	912	913	914	915	916	917	918	919	920	925
242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263
928	931	936	937	940	941	947	949	951	952	954	956	959	970	971	972	973	978	979	980	985	989
264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285

Implementing Binary Search

- The following method implements the binary search algorithm for an integer array.

```
private int binarySearch(int key, int[] array) {
   int lh = 0;
   int rh = array.length - 1;
   while (lh <= rh) {
      int mid = (lh + rh) / 2;
      if (key == array[mid]) return mid;
      if (key < array[mid]) {
         rh = mid - 1;
      } else {
         lh = mid + 1;
      }
   }
   return -1;
}
```

- The text contains a similar implementation of **`binarySearch`** that operates on strings. The algorithm is the same.

Efficiency of Linear Search

- As the area code example makes clear, the running time of the linear search algorithm depends on the size of the array.
- The idea that the time required to search a list of values depends on how many values there are is not at all surprising. The running time of most algorithms depends on the size of the problem to which that algorithm is applied.
- In many applications, it is easy to come up with a numeric value that specifies the problem size, which is generally denoted by the letter N. For most array applications, the problem size is simply the size of the array.
- In the worst case—which occurs when the value you're searching for comes at the end of the array or does not appear at all—linear search requires N steps. On average, it takes approximately half that time.

Efficiency of Binary Search

- The running time of binary search also depends on the number of elements, but in a profoundly different way.
- On each step in the process, the binary search algorithm rules out half of the remaining possibilities. In the worst case, the number of steps required is equal to the number of times you can divide the original size of the array in half until there is only one element remaining. In other words, what you need to find is the value of k that satisfies the following equation:

$$1 = N / \underbrace{2 / 2 / 2 / 2 \cdots / 2}_{k \text{ times}}$$

- You can simplify this formula using basic mathematics:

$$1 = N / 2^k$$
$$2^k = N$$
$$k = \log_2 N$$

Comparing Search Efficiencies

- The difference in the number of steps required for the two search algorithms is illustrated by the following table, which compares the values of N and the closest integer to $\log_2 N$:

N	$\log_2 N$
10	3
100	7
1000	10
1,000,000	20
1,000,000,000	30

- For large values of N, the difference in the number of steps required is enormous. If you had to search through a list of a million elements, binary search would run 50,000 times faster than linear search. If there were a billion elements, that factor would grow to 33,000,000.

Sorting

- Binary search works only on arrays in which the elements are arranged in order. The process of putting the elements of an array in order is called ***sorting***.
- There are many algorithms that one can use to sort an array. As with searching, these algorithms can vary substantially in their efficiency, particularly as the arrays become large.
- Of all the algorithms presented in this text, sorting is by far the most important in terms of its practical applications. Alphabetizing a telephone directory, arranging library records by catalogue number, and organizing a bulk mailing by ZIP code are all examples of sorting that involve reasonably large collections of data.

The Selection Sort Algorithm

- Of the many sorting algorithms, the easiest one to describe is ***selection sort***, which is implemented by the following code:

```
private void sort(int[] array) {
   for (int lh = 0; lh < array.length; lh++) {
      int rh = findSmallest(array, lh, array.length);
      swapElements(array, lh, rh);
   }
}
```

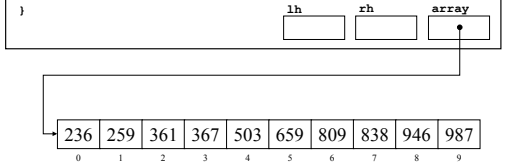
The variables **lh** and **rh** indicate the positions of the left and right hands if you were to carry out this process manually. The left hand points to each position in turn; the right hand points to the smallest value in the rest of the array.

- The method **findSmallest(***array***,** p_1**,** p_2**)** returns the index of the smallest value in the array from position p_1 up to but not including p_2. The method **swapElements(***array***,** p_1**,** p_2**)** exchanges the elements at the specified positions.

Simulating Selection Sort

```
public void run() {
   int[] test = { 809, 503, 946, 367, 987, 838, 259, 236, 659, 361 };
   sort(test);
}

private void sort(int[] array) {
   for ( int lh = 0 ; lh < array.length ; lh++ ) {
      int rh = findSmallest(array, lh, array.length);
      swapElements(array, lh, rh);
   }
}
```



The Efficiency of Selection Sort

- Chapter 12 includes a table of actual running times for the selection algorithm for arrays of varying sizes.
- Another way to estimate efficiency is to count how many time the most frequent operation is executed. In selection sort, this operation is the body of loop in **findSmallest**. The number of cycles in this loop changes as the algorithm proceeds:

  N values are considered on the first call to **findSmallest**.
  $N - 1$ values are considered on the second call.
  $N - 2$ values are considered on the third call, and so on.

- In mathematical notation, the number of values considered in **findSmallest** can be expressed as a summation, which can then be transformed into a simple formula:

$$1 + 2 + 3 + \cdots + (N-1) + N = \sum_{i=1}^{N} i = \frac{N \times (N+1)}{2}$$

Quadratic Growth

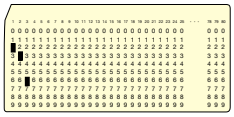
- The following table shows the value of $\frac{N \times (N+1)}{2}$ for various values of N:

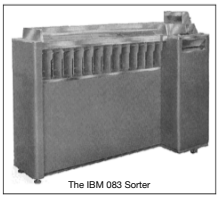
N	$\frac{N \times (N+1)}{2}$
10	55
100	5050
1000	500,500
10000	50,005,000

- The growth pattern in the right column is similar to that of the measured running time of the selection sort algorithm. As the value of N increases by a factor of 10, the value of $\frac{N \times (N+1)}{2}$ increases by a factor of around 100, which is 10^2. Algorithms whose running times increase in proportion to the square of the problem size are said to be ***quadratic***.

Sorting Punched Cards

From the 1880 census onward, information was often stored on punched cards like the one shown at the right, in which the number 236 has been punched in the first three columns.





The IBM 083 Sorter

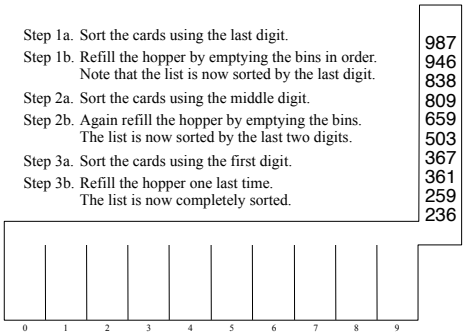
Computer companies built machines to sort stacks of punched cards, such as the IBM 083 sorter on the left. The stack of cards was loaded in a large hopper at the right end of the machine, and the cards would then be distributed into the various bins on the front of the sorter according to what value was punched in a particular column.

The Radix Sort Algorithm

- Although you will learn more efficient sorting algorithms if you go on to CS106B, the IBM 083 sorter does a much better job by using an algorithm called ***radix sort*** and consists of the following steps:
  1. Set the machine so that it sorts on the *last* digit of the number.
  2. Put the entire stack of cards in the hopper.
  3. Run the machine so that the cards are distributed into the bins.
  4. Put the cards from the bins back in the hopper, making sure that the cards from the 0 bin are on the bottom, the cards from the 1 bin come on top of those, and so on.
  5. Reset the machine so that it sorts on the preceding digit.
  6. Repeat steps 3 through 5 until all the digits are processed.
- The next slide illustrates this process for a set of three-digit numbers.

Simulating Radix Sort

Step 1a. Sort the cards using the last digit.
Step 1b. Refill the hopper by emptying the bins in order.
Note that the list is now sorted by the last digit.
Step 2a. Sort the cards using the middle digit.
Step 2b. Again refill the hopper by emptying the bins.
The list is now sorted by the last two digits.
Step 3a. Sort the cards using the first digit.
Step 3b. Refill the hopper one last time.
The list is now completely sorted.



Comparing N^2 and $N \log N$

- Like most of the algorithms you will encounter in CS106B, radix sort runs in time proportional to N times the log of N.
- The difference between N^2 and $N \log N$ can be enormous for large values of N, as shown in this table:

N	N^2	$N \log N$
10	100	33
100	10,000	664
1,000	1,000,000	9,966
10,000	100,000,000	132,877
100,000	10,000,000,000	1,660,964
1,000,000	1,000,000,000,000	19,931,569