5 Sorting

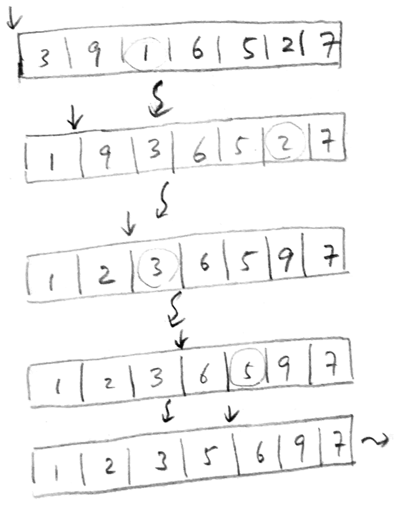
5.1 Introduction

[[A simple introduction to sorting and four different sorting methods: selection sort, insertion sort, quicksort, and mergesort.]]

Sorting is one of the commonest things computers do, usually employed as a part of bigger tasks. For example, once we have a sorted list, we can perform a binary search. We can do it quite efficiently in $O(n \log n)$ time. And we have a pretty nice story to tell about its computational complexity.

Let's start with the easiest ideas.

Selection Sort Can be done in-place in $O(n^2)$ time.



Choose the smallest among the remaing set

Keep repeating

For an input of size N, the number of comparisons is

$$= 1 + 2 + \cdots + N - 1 = N(N-1)/2 = O(N^2),$$

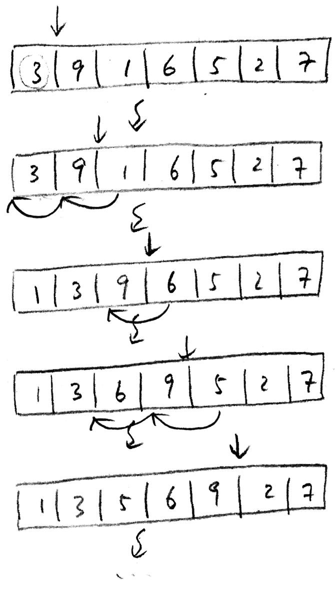
and the number of exchanges is

$$\leq N - 1 = O(N).$$

Insertion Sort An analogy of human-sorting of cards or exam papers. The same list above can be sorted by taking one by one and putting into an appropriate place of a seperate (and increasing) sorted list:

$$\langle 3 \rangle \rightarrow \langle 3, 9 \rangle \rightarrow \langle 1, 3, 9 \rangle \rightarrow \langle 1, 3, 6, 9 \rangle \rightarrow$$

We use two lists here, the given unsorted list we take items one by one from, and another sorted list that is increasing as we proceed. Can we use this idea to sort in-place, that is without using a separate list? Try this:



For an input of size N, the number of comparisons is

$$\leq 1 + 2 + \cdots + N - 1 = N(N-1)/2 = O(N^2),$$

and the number of exchanges is also

$$\leq 1 + 2 + \cdots + N - 1 = N(N-1)/2 = O(N^2).$$

Complexity of Sorting It is easy to see that these two methods takes $O(n^2)$. But we already know an $O(n \log n)$ time sorting algorithm.

(1) Put the list into a heap. It takes n `push`'s and each `push` takes $O(\log n)$ steps. So, in total, it is done in $O(n \log n)$ steps. We can use a min-heap. But it doesn't really matter.

(2) Use n `pop`'s. Again, each `pop` takes $O(\log n)$ steps. So, in total, it takes $O(n \log n)$ steps.

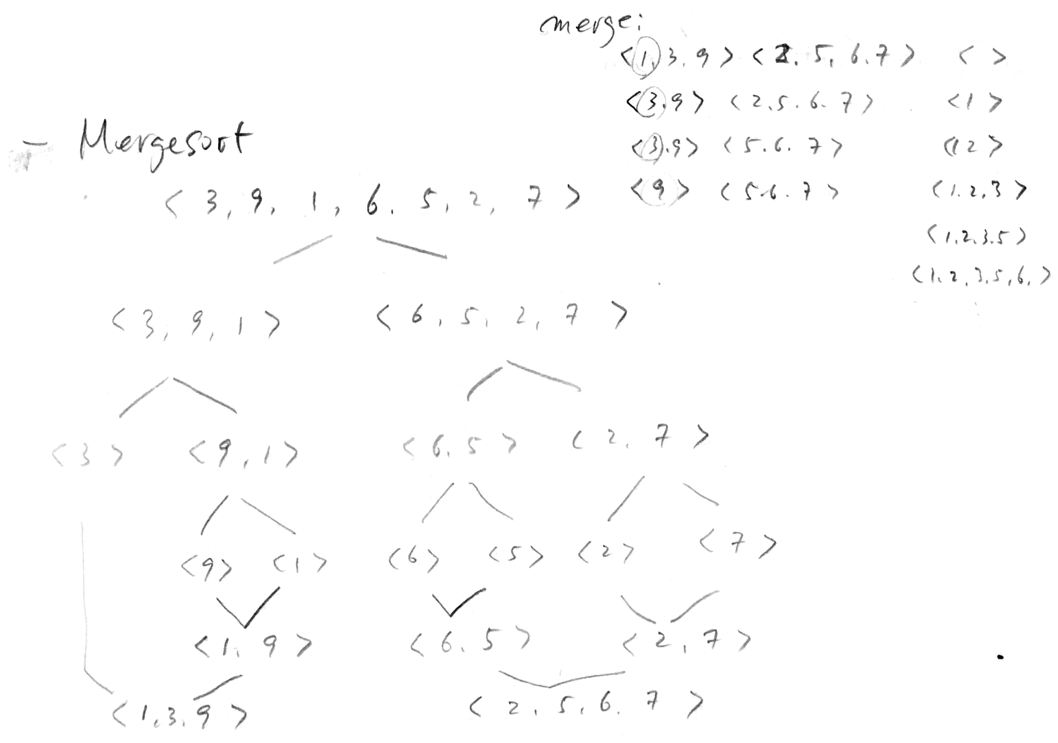
By (1) and (2), we accomplished a sorting in $O(n \log n)$ steps.

5.2 Mergesort

The (recursive version of) merge sort works as follows (with some abuse of notations and at a risk of inaccuracy....):

```
mergesort(A(l..r))
{
  if (l<r)
    m=(l+r)/2
    mergesort(A(l..m))
    mergesort(A((m+1)..r))
    merge(A(l..m),A((m+1)..r))
}
```

Here, `A(l..r)` means a sublist `(A(l), ..., A(r))` of a list `A`. Note that this is a recursive function. After the two sublists are sorted by recursive calls, they are `merge`d. Here, `merge()` combines two *sorted* lists and produces a single sorted list. In a sense, `merge()` is the essential part of mergesort, and it is where the actual sorting occurs. If there is another part, it is, of course, recursive *halving*. Here is an illustration of how mergesort works:



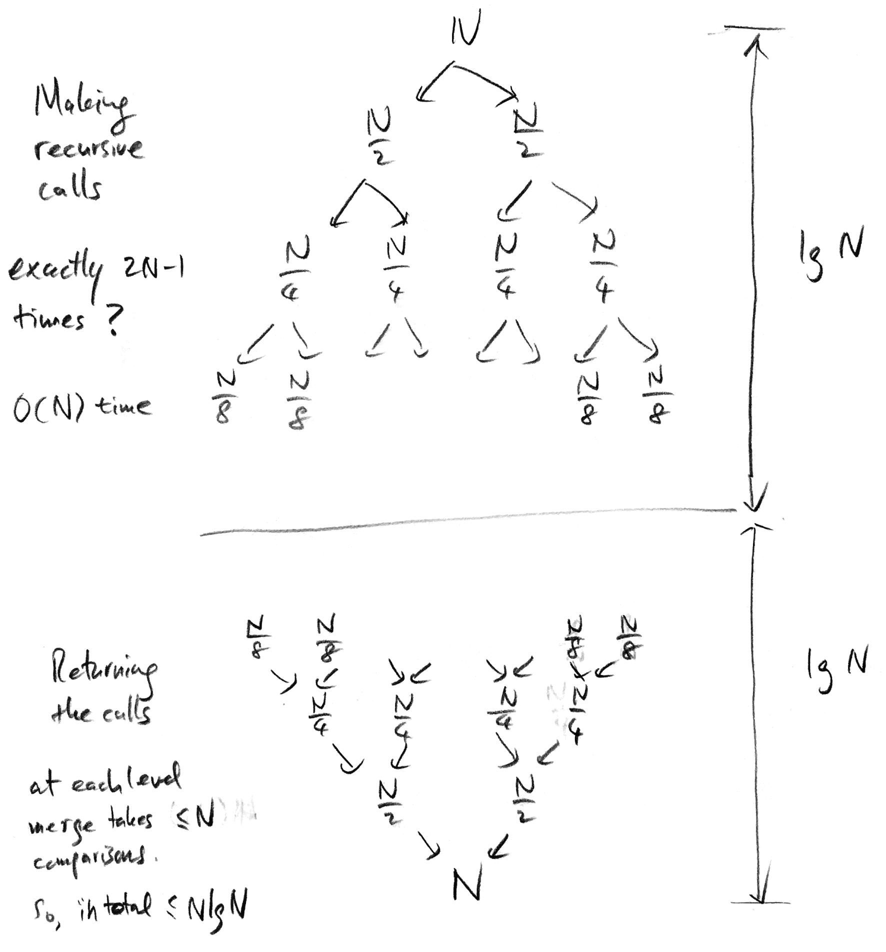
Now, let us see how we can make `merge()` work. Since it is the essential part, we need to make it efficient. Assume that the sublists `A(l..m)` and `A((m+1)..r)` are sorted. In case our list is an array, it is difficult to merge *in-place*. (We will see why.) So, we use a buffer `B(l..r)`.

```
merge(A(l..m..r))
{
  copy A(l..r) into B(l..r)
  i <- l
  j <- m+1
  k <- l     // index for recollecting into A
  while (i<=m and j<=r)
    if (B(i)<=B(j))
      A(k++) <- B(i++)
    else
      A(k++) <- B(j++)
  if (i>m)  // taking care of the remaining part
```

```
      while (j<=r)
        A(k++) <- B(j++)
    else
      while (i<=m)
        A(k++) <- B(i++)
  }
```

Observe that merging sorted *arrays* of sizes m and n takes at least $\min(m, n)$ and at most $m + n - 1$ comparisons. Moving data between the array and the buffer occurs $2(m + n)$ times. So it takes $O(m + n)$ time. If we work with *linked lists*, the comparisons occur the same number of times, but the data movement (actually just changing the links) can be done in exactly $\min(m + n)$, for each comparison, plus once, for taking care of the remainder. Moreover, we do not need the buffer! But merge sort with linked list is awkard to do with recursion and need some more work. (See [TAOCP vol.2] for details.)

Analysis of Recursive Merge Sort Making recursive calls is a fancy way to *divide* the array and put them together again. For an input of size N, it takes exactly $2N - 1$ recursive calls. The recursion tree is an extended binary tree, whose leaves are exactly the recursive calls with argument of size 1, therefore that does not spawn a subtree. So, they are exactly the empty binary trees. There are N leaves, and there are $N - 1$ (internal) nodes above them.



5.2.1 Mergesort Implementation

The following is an implementation of top-down, or recursive mergesort.

```
// Sort N = a few million 32-bit integers using recursive mergesort.
#include <stdio.h>
#define N 1000000  // Say we have one million inputs.

void merge(int arr[], int l, int m, int r) // l<m<r
{
  int b[N+2];  // buffer to store two sublists.
  int i=l;
  int j=m+1;
  int k=l;

  // copy sublists to buffer
  for(i=l; i<=m; i++)
```

```
    b[i]=arr[i];
  for(j=m+1; j<=r; j++)
    b[j]=arr[j];

  i=l; j=m+1;  // reset indices.

  // merge sublists in buffer into arr[]
  while (i<=m && j<=r) {
    if (b[i]<=b[j])
      arr[k++]=b[i++];
    else
      arr[k++]=b[j++];
  }
  if (i>m)
    while (j<=r)
      arr[k++]=b[j++];
  else
    while (i<=m)
      arr[k++]=b[i++];
}

void mergesort(int arr[], int l, int r)
{
  int m;

  if (l < r) {
    m = (l+r)/2;
    mergesort(arr, l, m);
    mergesort(arr, m+1, r);
    merge(arr, l, m, r);
  }
}

int main()
{
  int i;
  int data0[N];

  // read one million data
  for(i=0; i<N; i++) {
    scanf("%d", &data0[i]); // read the input
  }

  mergesort(data0,0,N-1);

  // output
  for(i=0; i<N; i++)
    printf("%d ", data0[i]);
}
```

5.3 Quicksort

Quicksort is probably the most commonly used sorting algorithm, and that's for a good reason. We will see that, in the worst case, it will take $O(n^2)$ time. But mostly it takes $O(n \log n)$ time and usually, in practice, quicksort is faster than most other sorting algorithms.

Quicksort is, again, a divide-and-conquer algorithm, and can be described nicely as a recursion:

```
quicksort(A(l..r))
{
  if(l<r)
    p <- partition(A(l..r))
    quicksort(A(l..p-1))
    quicksort(A(p+1..r))
}
```

As `merge()` does an actual work with mergesort, `partition()` is an essential part of quicksort. As we will see below, the partition requires random accesses to the elements of the list. So, quicksort is not appropriate for linked lists. But, instead, probably at the expense of random access, it can be done in-place. Assume the list is an array. Given below is quite an efficient implementation of quicksort on an integer array.

5.3.1 Quicksort Implementation

```
// Sort N = a few million 32-bit integers using recursive quicksort.
#include <stdio.h>
#include <limits.h>  // to use INT_MAX as a sentinel
#define N 1000000  // Say we have one million inputs.

void swap(int *x, int *y)
{
  int temp=*x;
  *x=*y;
  *y=temp;
}

int partition(int arr[], int l, int r)
{
  int x=arr[l];  // pivot!
  int i=l+1;
  int j=r;

  while (1) {
  //while (arr[i] <= x && i<=r) i++;
  //while (arr[j] > x && j>=l) j--;
  // Need a bound check; consider the case pivot is the largest.
    while (arr[i] <= x) i++; // avoid bound check by putting sentinel
    while (arr[j] > x) j--; // bound check is unnecessary; arr[l] is sentinel
    if (i<j) {
      swap(&arr[i],&arr[j]);
      i++; j--;  //  // new start points for probing
    }
    else break;
  }
  swap(&arr[l],&arr[j]);
  return j;
}

void myQuicksort(int arr[], int l, int r)
{
  int p;

  if (l < r) {
  p = partition(arr, l, r);
  myQuicksort(arr, l, p-1);
  myQuicksort(arr, p+1, r);
  }
}

int main()
{
  int i;
  int data[N+1];  // Plus one for sentinel.

  // read one million data
  for(i=0; i<N; i++) {
    scanf("%d", &data[i]); // read from stdin
  }

  data[i] = INT_MAX;  //**IMPORTANT** sentinel at the end
  // Why is this enough?  Except for the sublist at the end,
  // all sublists to be partitioned already has a sentinel :)

  myQuicksort(data,0,N-1);

  // output
  for(i=0; i<N; i++)
    printf("%d ", data[i]);
}
```

On one million 32-bit random integers, as the program was written for, this program is slightly faster than the library function `qsort()`. It is probably because `qsort()` needs to be used with a comparison function and thus have function call overhead, while the above implementation directly uses integer comparison.

Suppose that it was used as follows:

```
PROMPT> myQuicksort < onemillion.in > onemillion.out
```

The program finishes in less than one second. And then, run the same program on the sorted data `onemillion.out`.

```
PROMPT> myQuicksort < onemillion.out > test.out
```

As expected, the program does not finish quickly. In fact, it runs for several minutes, and then gives up with a "Segmentation fault," because one million recursive calls need to be made to finish. Probably, several tens of thousands recursive calls are made before the stack is overflowed, which is 8MB on Mac OS X by default.

An analysis With N distinct items, the `partition()` takes exactly $N-1$ comparisons. The number of comparisons for quicksort is, therefore, ... [[best case: $O(n \log n)$ and the worst case: $O(n^2)$. Then, what is the typical case or the average cost? See the following homework]]