CS 137 Week 8

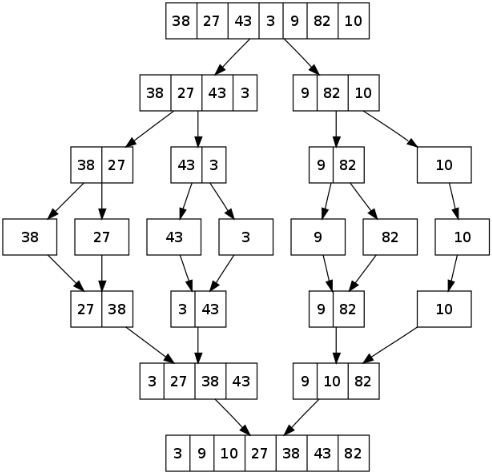
Merge Sort, Quick Sort, Binary Search

November 6th, 2017

This Week

- We're going to see two more complicated sorting algorithms that will be our first introduction to $O(n \log n)$ sorting algorithms.
- The first of which is Merge Sort.
- Basic Idea:
    1. Divide array in half
    2. Sort each half recursively
    3. Merge the results

Example



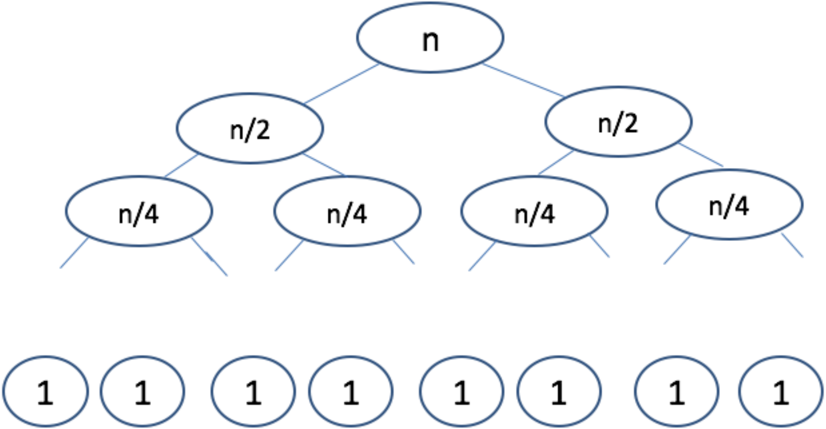
https://upload.wikimedia.org/wikipedia/commons/e/e6/Merge_sort_algorithm_diagram.svg

Merge Sort

```
void sort(int a[], int n) {
  int *t = malloc(n*sizeof(a[0]));
  assert(t);
  merge_sort(a, t, n);
  free(t);
}
int main (void){
  int a[] = {-10,2,14,-7,11,38};
  int n = sizeof(a)/sizeof(a[0]);
  sort(a,n);
  for (int i = 0; i < n; i++) {
      printf("%d, ", a[i]);
  }
  printf("\n");
  return 0;
}
```

```c
void merge_sort (int a[], int t[], int n) {
  if (n <= 1) return;
  int middle = n / 2;
  int *lower = a;
  int *upper = a + middle;
  merge_sort(lower, t, middle);
  merge_sort(upper, t, n - middle);
  int i = 0;      // lower index
  int j = middle; // upper index
  int k = 0;      // temp  index
  while (i < middle && j < n) {
      if (a[i] <= a[j]) t[k++] = a[i++];
      else t[k++] = a[j++];
  }
  while (i < middle) t[k++] = a[i++];
  while (j < n) t[k++] = a[j++];
  for (i = 0; i < n; i++) a[i] = t[i];
}
```

Runtime of Merge Sort



Analysis

How much work is done by each instance?

- Two function calls of $O(1)$
- Copy the left and right into `t[]` is $O(k)$ where k is the size of the current instance
- Copy `t[]` into `a[]` which is also $O(k)$.
- Therefore, each instance of a merge sort is $O(k)$.

Continuing

So each instance is $O(k)$ but how many instances are there at each level?

- The number of bubbles per row is n/k. This follows since after $k = n/2^\ell$ halves, we've created 2^ℓ bubbles and this is n/k.
- Thus, for each level, the total amount of work done is $O(n/k \cdot k) = O(n)$.

Wrapping Up

- Finally, how many levels are there?
- To answer this, we are looking for a number m such that $\frac{n}{2^m} = 1$.
- Solving gives $m = \log_2(n)$.
- Hence, the total time is $O(n \log n)$.
- This analysis applies for the best, worst and average cases!

Quick Sort

- Created by Tony Hoare in 1959 (or 1960)
- Basic Idea:
  1. Pick a pivot element p in the array
  2. Partition the array into

$< p$	p	$\geq p$

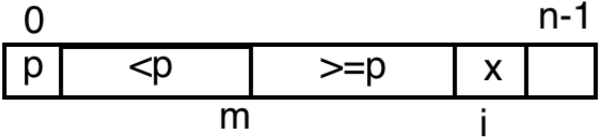
  3. Recursively sort the two partitions.
- Key benefit: No temporary array!

Tricky Point

- How do we pick the pivot and partition?
- We will discuss two such ways, the Lomuto Partition and choosing a median of medians.
- The Lomuto Partition is usually easier to implement but doesn't preform as well in the worst case.
- The median of medians enhancement vastly improves the worst case runtime
- Note that Hoare's original partitioning scheme has similar runtimes to Lomuto's but is slightly harder to implement so we won't do so.

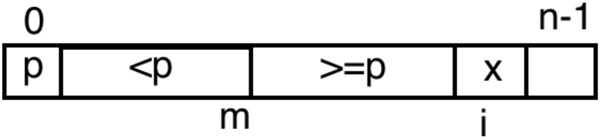
Lomuto Partition

- Swaps left to right.
- Pivot is first element (Could by last element with modifications below if desired).
- Have two index variables:
  1. `m` which represents the last index in the partition $< p$
  2. `i` which represents the first index of the unpartitioned part.
  3. In other words



So elements with indices between 1 and m inclusive are $< p$ and elements with indices

Lomuto Partition Continued

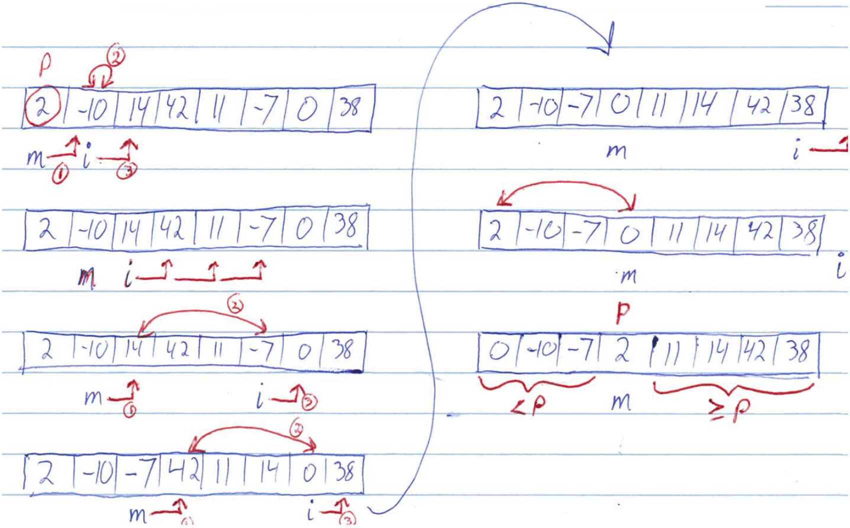


There are two cases:

- If $x < p$, then increment `m` and swap `a[m]` with `a[i]` then increment `i`
- If $x \geq p$, then just increment i.
- End case: When $i = n$, then we swap `a[0]` and `a[m]` so that the partition is in the middle.

Let's see this in action

Quick Sort Example



Quick Sort Main

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
// Code here is on next slides
int main(void){
  int a[] = {-10,2,14,-7,11,38};
  const int n = sizeof(a)/sizeof(a[0]);
  quick_sort(a,n);
  for (int i = 0; i < n; i++) {
      printf("%d, ", a[i]);
  }
  printf("\n");
  return 0;
}
```

Swapping

```
void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}
```

Quick Sort

```c
void quick_sort(int a[], int n) {
  if (n <= 1) return;
  int m = 0;
  for (int i = 1; i < n; i++) {
    if (a[i] < a[0]) {
      m++;
      swap(&a[m],&a[i]);
    }
  }
  // put pivot between partitions
  // i.e, swap a[0] and a[m]
  swap(&a[0],&a[m]);
  quick_sort(a, m);
  quick_sort(a + m + 1, n - m - 1);
}
```

Time Complexity Analysis

Best Case:

- Ideally, each partition is split into (roughly) equal halves.
- Going down the recursion tree, we notice that the analysis here is almost identical to merge sort.
- There are $\log n$ levels and at each level we have $O(n)$ work.
- Therefore, in the best case, the runtime is $O(n \log n)$.

Time Complexity Analysis

Average Case:

- Again this isn't quite easy to quantify but we'll suppose at each level, the pivot is between the 25th and 75th percentiles.
- Then, in this case, the worst partition is 3:1.
- Now, there are $\log_{4/3} n$ levels (solve $(3/4)^m n = 1$) and at each level we still have $O(n)$ work.
- Therefore, in the average case, the runtime is $O(n \log n)$.

Time Complexity Analysis

Worst Case:

- Worst case turns out to be very poor.
- What if the array were sorted (or reverse sorted)?
- Then the array is always partitioned into an array of size 1 and an array of size `len-1`.
- At each level we still have $O(n)$ work.
- Unfortunately, there are now n levels for a total runtime of $O(n^2)$.
- To be slightly more accurate, note that at each level, we have a constant times

$$(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}=O(n^2)$$

amount of work.

Well if that's the case...

- ... then why is quick sort one of more frequently used sorting algorithms?
- One of the reasons why is that we have other schemes that can help ensure that even in the worst case, we get $O(n \log n)$ as our runtime.
- The key idea is picking our pivot intelligently.
- Turns out while this does theoretically reduces our worst case runtime, often in practice this isn't done because it increases our overhead of choosing a pivot.

Idea

- What we'll do is group the array into groups of five (a total of $n/5$ such groups)
- Then we'll take the median of each of those groups
- Now of these $n/5$ medians, repeat until you eventually find the median of these medians. Call this k
- Thus, with this k, we know that k is bigger than $n/10$ medians and each of those medians was at least the size of 3 other numbers (including itself) and so in total, we know that k is bigger than $3n/10$ of the numbers and similarly that it is smaller than $7n/10$ of the numbers.
- For a more formal proof take CS 341 (Algorithms)!

Picture for 25 elements

a	b	c	d	e
f	g	h	i	j
l	m	k	n	o
p	q	r	s	t
u	v	w	x	y

Space Requirements for Quick Sort

- Partition uses only a constant number of variables (`m`, `i` and possibly swapping)
- However Recursive calls add to the stack.
- In the best case, we have 50/50 splits and in this case, the stack only has size $\log n$
- In the worst case, we have 1 and $n - 1$ splits (where n changes with the length of the array we are considering) and so we have a stack with size n.

Picture of the splits

Best Case

```
quicksort(a,1)
⋮
quicksort(a,n/4)
quicksort(a,n/2)
quicksort(a,n)
```

stack

Worst Case

```
quicksort(a,1)
⋮
quicksort(a,n-5)
quicksort(a,n-4)
quicksort(a,n-3)
quicksort(a,n-2)
quicksort(a,n-1)
quicksort(a,n)
```

stack

Optimizations

Tail Recursion

This is when the recursive call is the last action of the function

For example,

```
void quicksort(int a[], int n){
  //Commands here... no recursion
  quicksort(a+m+1,n-m-1);
}
```

Tail Call Elimination

- When the recursive call returns, its return is simply passed on.
- Therefore, the activation record of the caller can be reused by the callee and thus the stack doesn't grow.
- This is guaranteed by some languages like Scheme
- It can be enabled in gcc by using the -O2 optimization flag.
- With this and sorting the smallest partition first, the stack depth in the worst case space complexity of quick sort is $O(\log n)$.

Built in Sorting

- In practice, people almost never create their own sorting algorithms as there is usually one built into the language already.
- For C, `<stdlib.h>` contains a library function called `qsort` (Note: This isn't necessarily quick sort!)

```
void qsort(void *base, size_t n, size_t,size,
 int  (*compare)(const void *a, const void *b));
```

Description of Parameters

- `base` is the beginning of the array
- `n` is the length of the array
- `size` is the size of a byte in the array
- `*compare` is a function pointer to a comparison criteria.

Compare Function

```c
#include <stdlib.h>
int compare(const void *a, const void *b) {
  int p = *(int *)a;
  int q = *(int *)b;
  if (p < q) return -1;
  else if (p == q) return 0;
  else return 1;
}
void sort(int a[], int n) {
  qsort(a,n,sizeof(int),compare);
}
```

Alternatively

```
int compare(const void *a, const void *b) {
  int p = *(int *)a;
  int q = *(int *)b;
  return (p < q) ? -1 : ((q > p) ? 1 : 0);
}
```

Binary Search

- Recall we started with linear searching which we said had a $O(n)$ runtime.
- If we already have a sorted array, linear searching seems like we're not effectively using our information.
- We'll discuss binary searching which will allow us to search through a sorted array.

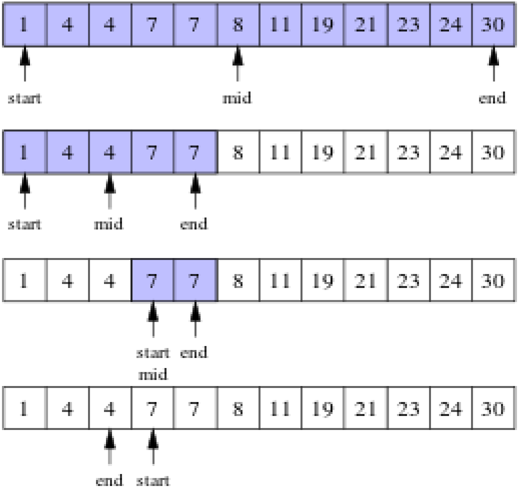
Binary Searching

Basic Idea

- Check the middle element `a[m]`
- If we match with `value`, return this element
- If `a[m] > value` then search the lower half
- If `a[m] < value` then search the upper half
- Stop when `lo > hi` where lo is the lower index and hi is the upper index (start and end in the next graphic)

Binary Search

Let's find 6 in the following sorted array



https://puzzle.ics.hut.fi/ICS-A1120/2016/notes/round-efficiency--binarysearch.html

Binary Search

```c
#include <stdio.h>
int search(int a[], int n, int value) {
  int lo = 0, hi = n-1;
  while (hi >= lo) {
    //Note int m = (hi +lo)/2 is equivalent
    //but may overflow.
    //Same with (hi+lo)>>1;
    int m = lo+(hi-lo)/2;
    if (a[m] == value) return m;
    if (a[m] < value) lo = m+1;
    if (a[m] > value) hi = m-1;
  }
  return -1;
}
```

Binary Searching

```
int main() {
  int a[] = {-10,7,0,2,11,14,38,42};
  const int n = sizeof(a)/sizeof(a[0]);
  printf("%d\n", search(a,n,10));
  printf("%d\n", search(a,n,2));
  printf("%d\n", search(a,n,42));
  return 0;
}
```

Tracing

Let's trace the code with `value = 10, 11, -100`

Time Complexity of Binary Search

Worst Case (and Average Case) Analysis

- If we don't find the element, then at each step we search only in half as many elements as before and searching takes only $O(1)$ work.
- We can cut the array in half a total of $O(\log n)$ times.
- Thus, the total runtime is $O(\log n)$.

Best Case Analysis

- We find the element immediately and return taking only $O(1)$ time.

Note this is extremely fast - even with 1 billion integers, we only need at worst 30 probes.

Sorting Summary

Selection sort

- Find the smallest and swap with the first
- Runtime for best, average and worst case: $O(n^2)$

Insertion sort

- Find where element i goes and shift the rest to make room for element i.
- Runtime for best case is $O(n)$ and for average and worst case: $O(n^2)$

Sorting Summary

Merge sort

- Divide and conquer - Split in halves, recurse then merge.
- Runtime for best, average and worst case: $O(n \log n)$ but has $O(n)$ extra space.

Quick Sort

- Pick pivot, split elements into smaller and large than pivot, repeat.
- Runtime for best, average and worst case: $O(n \log n)$

Searching Summary

Linear Search

- Scan one by one until found
- Runtime for best case is O(1) and for, average and worst case: $O(n)$

Binary Search

- Probe middle and repeat (requires sorted array)
- Runtime for best case is O(1) and for, average and worst case: $O(\log n)$