Introduction to Algorithms

DEPARTMENT OF
COMPUTER SCIENCE

Neelam Akula

Spring 2021

Last Updated: March 24, 2021

Contents

1 Maximum Subarray Problem

Given an array of numbers (either positive or negative), we wish to calculate the subset of consecutive numbers whose sum is largest.

$$A: \boxed{3}\boxed{2}\boxed{-4}\boxed{-5}\underbrace{\boxed{6}\boxed{1}\boxed{-3}\boxed{7}}_{\text{sums to 11}}\boxed{-8}\boxed{2}$$

In this example, we see that the maximum sum is 11 with the indecies of the subarray being $A[5] : A[8]$.

1.1 Cubic Time

Pseudocode

Algorithm 1 Maximum Continguous Sum (*Cubic*)

```
M ← 0
for i = 1 to n do
    for j = i to n do
        S ← 0
        for k = i to j do
            S ← S + A[k]
        end for
        M ← MAX(M, S)
    end for
end for
```

Analysis

From our algorithm above we can get the following analysis for it's complexity.

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j}1 &= \sum_{i=1}^{n}\sum_{j=i}^{n} j-i+1\\ &= \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} j\\ &= \sum_{i=1}^{n}\frac{(n-i+1)(n-i+2)}{2}\\ &= \frac{1}{2}\sum_{i=1}^{n}(n-i+1)(n-i+2)\\ &= \frac{1}{2}\sum_{n-i+1=1}^{n} i(i+1)\\ &= \frac{1}{2}\frac{n(n+1)(n+2)}{3}\end{aligned}$$

From this, we can see this first version of our algorithm is $\Theta(n^3)$ or cubic time.

1.2 Quadratic Time

Pseudocode

Algorithm 1 Maximum Continguous Sum (*Quadratic*)

```
M ← 0
for i = 1 to n do
    S ← 0
    for j = i to n do
        S ← S + A[j]
        M ← MAX(M, S)
    end for
end for
```

Analysis

We then use sums to analyze this algorithm.

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=i}^{n} 1 &= \sum_{i=1}^{n} n-i+1\\ &= \sum_{i=1}^{n} i\\ &= \frac{n(n+1)}{2}\end{aligned}$$

So this more efficient version is $\Theta(n^2)$ or quadratic time.

1.3 Linear Time

Using dynamic programming we can improve on this algorithm even further. Conceptually, the maximum sum is just the previous maximum sum plus the current value.

Pseudocode

Algorithm 1 Maximum Continguous Sum (*Linear*)

$M \leftarrow 0$
$S \leftarrow 0$
for $i = 1$ **to** n **do**
  $S \leftarrow \textsc{max}(S + A[i], 0)$
  $M \leftarrow \textsc{max}(M, S)$
end for

Analysis

It follows that this algorithm is $\Theta(n)$ or linear time. Correctness of this algorithm stems from proof by induction on $S \leftarrow \textsc{max}(S + A[i], 0)$, as this is the loop invariant.

2 Bubble Sort

2.1 Pseudocode

Algorithm 2 Bubble Sort

for $i = n$ **down to** 2 **do**
    for $j = 1$ **to** $i - 1$ **do**
        if $A[j] > A[j+1]$ **then**
            $A[j] \leftrightarrow A[j+1]$
        end if
    end for
end for

2.2 Analysis of Comparisons

Bubble Sort is designed in such a manner such that the state of the array to be listed does *not* change the number of comparisons it makes.

$$\sum_{i=2}^{n}\sum_{j=1}^{i-1} 1 = \sum_{i=2}^{n} i - 1$$
$$= \sum_{i=1}^{n-1} i$$
$$= \frac{(n-1)n}{2} = \binom{n}{2}$$

2.3 Analysis of Exchanges

Worst Case

Worst case is when the list is reverse sorted, there will be the same number of exchanges as comparisons.

$$\frac{(n-1)n}{2}$$

Best Case

Best case is when the list is already sorted, in which there will be zero exchanges.

Average Case

To find the average case we must count the transpositions (two elements that are out of order related to one another). In best case there are no transpositions, and in worst case there are $\frac{(n-1)n}{2}$ transpositions. In a randomly permuated array each element is equally likely to be out of order so the total number of average case exchanges is half the comparisons.

$$\frac{1}{2} \cdot \frac{(n-1)n}{2} = \frac{(n-1)n}{4}$$

3 Insertion Sort with Sentinel

3.1 Pseudocode

Algorithm 3 Insertion Sort with Sentinel

```
A[0] ← −∞
for i = 2 to n do
    t ← A[i]
    j ← i − 1
    while t < A[j] do
        A[j + 1] ← A[j]
        j ← j − 1
    end while
    A[j + 1] ← t
end for
```

3.2 Analysis of Comparisons

Worst Case

Worst case is when the array is reverse sorted, and every element must be moved. The while loop always decrements j to zero to compare against the sentinel value.

$$\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1 = \frac{(n+1)n}{2} - 2 = \frac{(n+2)(n-1)}{2}$$

Best Case

Best case is when the array is already sorted, there is only one comparison for each iteration of the for loop.

$$\sum_{i=2}^{n} 1 = (n-2) + 1 = n - 1$$

Average Case

For average case we have to determine the probability that a given element will move. So we want the expected value of $\sum_{x \in X} P(x)V(x)$, where $P(x)$ is the probability that an element will end up a location and $V(x)$ is the number of moves.

$$\begin{aligned}
\sum_{x\in X} P(x)V(x) &= \sum_{i=2}^{n}\sum_{j=1}^{i}\frac{1}{i}\cdot(i-j+1) = \sum_{i=2}^{n}\frac{1}{i}\sum_{j=1}^{i}(i-j+1) \\
&= \sum_{i=2}^{n}\frac{1}{i}\sum_{j=1}^{i} j = \sum_{i=2}^{n}\frac{1}{i}\cdot\frac{(i+1)i}{2} \\
&= \sum_{i=2}^{n}\frac{i+1}{2} = \frac{1}{2}\sum_{i=2}^{n} i+1 \\
&= \frac{1}{2}\sum_{i=1}^{n} i-1+(n-1) \\
&= \frac{1}{2}\left(\frac{(n+1)n}{2}-1+\frac{(n-1)2}{2}\right) \\
&= \frac{(n+4)(n-1)}{4}
\end{aligned}$$

3.3 Analysis of Exchanges

Worst Case

Worst case is two moves for each iteration of outer loop plus worst case number of comparisons, minus the time when the comparison is false at the end. A shortcut of this is at each iteration we do one more move than comparison, so take the value we got above and add the number of loop iterations.

$$\frac{(n+2)(n-1)}{2}+n$$

Best Case

Best case is one initial move in the assignment on line 1 and then two moves during each iteration of the for loop. So we get,

$$1+2(n-1)=2n-1$$

Average Case

Average case is whenever there is a comparison there is a move except for the single instance in each iteration of when it evaluates to false. Thus the analysis method is the same as worst case.

$$\frac{(n+4)(n-1)}{4}+n$$

4 Insertion Sort without Sentinel

4.1 Pseudocode

Algorithm 4 Insertion Sort without Sentinel

```
for i = 2 to n do
    t ← A[i]
    j ← i − 1
    while j > 0 and A[j] > t do
        A[j + 1] ← A[j]
        j ← j − 1
    end while
    A[j + 1] ← t
end for
```

4.2 Analysis of Comparisons

Worst Case

Worst case is when the array is reverse sorted, and every ith iteration of the loop must compare against all previous $(i-1)$ elements.

$$\begin{aligned}\sum_{i=2}^{n}\sum_{j-1}^{i-1} 1 = \sum_{i=2}^{n}(i-1) &= \sum_{i=2}^{n} i - \sum_{i=2}^{n} 1\\ &= \frac{(n+1)n}{2} - (1-(n-1))\\ &= \frac{(n-1)n}{2}\end{aligned}$$

Best Case

Best case is when the array is already sorted, so there will just be 1 comparison per iteration of the outermost loop. (Same as Insertion Sort with Sentinel.)

$$\sum_{i=2}^{n} 1 = (n-2) + 1 = n - 1$$

Average Case

On average, the sentinel costs $\sum_{i=2}^{n} \frac{1}{i}$ comparisons. This is simply just the Harmonic Series. In other words, the sentinel costs $H_n - 1$ comparisons, so Insertion

Sort without Sentinel is

$$\frac{(n+4)(n-1)}{4} - (H_n - 1) \approx \frac{(n+4)(n-1)}{4} - \ln n$$

4.3 Analysis of Exchanges

Removing the sentinel adds no new exchanges so the best, worst, and average cases are all the same as Insertion Sort with Sentinel.

5 Selection Sort

5.1 Pseudocode

Algorithm 5 Selection Sort

```
for i = n down to 2 do
    k ← 1
    for j = 2 to i do
        if A[j] > A[k] then
            k ← j
        end if
    end for
    A[k] ↔ A[i]
end for
```

5.2 Analysis of Comparisons

The number of comparisons is constant regardless of the state of the array.

$$\sum_{i=2}^{n} \sum_{j=2}^{i} 1 = \sum_{i=2}^{n} (i-1) = \frac{(n-1)}{n}$$

5.3 Analysis of Exchanges

Selection sort only performs one exchange per each iteration of the outermost loop, so there is a total of $n-1$ exchanges.

6 Merge Sort

6.1 Pseudocode

Algorithm 6 Merge Sort

procedure MERGESORT(A, p, r)
    if $p < r$ **then**
        $q \leftarrow \lfloor (p + r)/2 \rfloor$
        MERGESORT(A, p, q)
        MERGESORT$(A, q + 1, r)$
        MERGE$(A, (p, q), (q + 1, r))$
    end if
end procedure

procedure MERGE$(A, (p, q), (q + 1, r))$
    copy (A, p, r) into (B, p, r)
    $i \leftarrow p$
    $j \leftarrow q + 1$
    $k \leftarrow p$
    while $i \leq q$ and $j \leq r$ **do**
        if $B[i] \leq B[j]$ **then**
            $A[k] \leftarrow B[i]$
            $i \leftarrow i + 1$
        else
            $A[k] \leftarrow B[j]$
            $j \leftarrow j + 1$
        end if
        $k \leftarrow k + 1$
    end while
    if $i > q$ **then**
        copy (B, j, r) into (A, k, r)
    else
        copy (B, i, q) into (A, k, r)
    end if
end procedure

6.2 Analysis of Merge

Equal Size

Best case, we only do n comparisons (this is when $A_n < B_1$ or $B_n < A_1$). Worst case is $2n - 1$ comparisons (this is when A_n and B_n are the two largest elements).

The average case is $2n - 2 + \frac{2}{n+1}$.

Differing Sizes

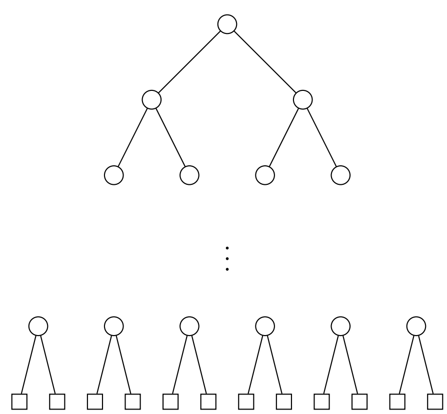
Let subarray A be of size m and subarray B be of size n where $m \leq n$. Best case is m comparisons (this is when $A_m < B_1$). Worst case is $m + n - 1$ (this is when A_m and B_n are the two largest elements). When m is much smaller than n there are better algorithms we can use. For $m = 1$ binary search gives $\approx \lg n$.

Notes

Merging is *not* in place. It can be implemented in place but those algorithms are not practical.

6.3 Analysis of Mergesort Comparisons

We will analyze mergesort using the tree method, along with the assumption that n is a power of 2.

Problem Size	Tree	Comparisons
n		$n-1$
$\frac{n}{2^1}$		$2\left(\frac{n}{2}-1\right)$
$\frac{n}{2^2}$		$4\left(\frac{n}{4}-1\right)$
$\vdots$	$\vdots$	$\vdots$
2		n
1		$0n$

We can notice that each level does exactly $2^k(\frac{n}{2^k} - 1)$ comparisons where k is the level of that branch, it follows then that since the height of the tree is $\lg n$, the total number of comparisons is

$$\begin{aligned}
\sum_{i=0}^{\lg n-1} 2^i \left(\frac{n}{2^i} - 1\right) &= \sum_{i=0}^{\lg n-1} \left(n - 2^i\right) \\
&= \sum_{i=0}^{\lg n-1} n - \sum_{i=0}^{\lg n-1} 2^i \\
&= (n \lg n) - \left(2^{\lg n-1+1} - 1\right) \\
&= (n \lg n) - (n-1) \\
&= n \lg n - n + 1
\end{aligned}$$

As a recurrence,

$$\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + (n-1) \\
&= 2T\left(\frac{n}{2}\right) + n - 1, \quad T(0) = T(1) = 0
\end{aligned}$$

7 Heap Sort

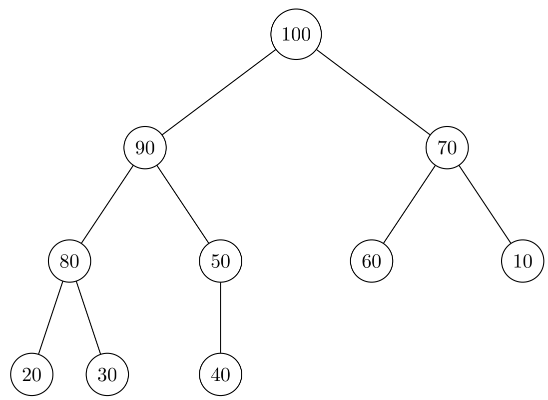
7.1 Pseudocode

Algorithm 7 Heap Sort

```
procedure HEAPSORT(A, n)                    ▷ A is a list and n is A's size
    for r = ⌊n/2⌋ down to 1 do                                ▷ Create Heap
        SIFT(r, n)
    end for

    for m = n down to 2 do                                    ▷ Finish Sort
        A[1] ↔ A[m]
        SIFT(1, m − 1)
    end for
end procedure

procedure SIFT(p, m)        ▷ p is the root and m is the size of the list
    c ← 2p
    while c ≤ m do
        if c < m then
            if A[c + 1] > A[c] then
                c ← c + 1
            end if
        end if

        if A[c] > A[p] then
            A[p] ↔ A[c]
            p ← c
            c ← 2p
        else
            exit while loop
        end if
    end while
end procedure
```

7.2 Create Heap

A Heap is a binary tree where every value is larger than its children. Equivalently its descendants. For the purposes of this class will we require that all binary trees are full binary trees.



The traditional way of creating a heap is to insert at the end of the array and sift up. Robert Floyd created a better algorithm for creating the heap. Treat the tree as a recursive heap: each parent is the parent of 2 heaps, and sift from the bottom up. Create heap on left, create heap on right, then sift root down, and move up a level.

Heap Creation Analysis

In a binary tree, most nodes are near the bottom, so when doing the bottom up technique, most of the work is done at the bottom. The number of comparisons can be calculated like so,

$$\begin{aligned} \frac{n}{2}\cdot 0 + \frac{n}{4}\cdot 2 + \frac{n}{8}\cdot 4 + \cdots = \\ = n\left[\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots\right] \end{aligned}$$

Note that $\left[\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots\right]$ can be written as so,

$$\begin{aligned} \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots &= 1 \\ \frac{1}{4} + \frac{1}{8} + \cdots &= \frac{1}{2} \\ \frac{1}{8} + \cdots &= \frac{1}{4} \end{aligned}$$

Which all together becomes

$$1 + \frac{1}{2} + \frac{1}{4} + \cdots = 2$$

So Heap creation does $2n$ comparisons.

7.3 Finish Sort

After heap creation we then sort it,

- Put root node at the bottom of the array (it must be the largest element) so it goes to end of the sorted array.
- Then take the bottom right hand leaf and move to a temporary space.
- Then sift, reordering the tree and put the temporary leaf in its proper spot.
- Repeat until all elements are sorted.

Heap Sort Analysis

Each level has two comparisons, child and temporary. There are $\approx \lg n$ levels, so the total comparisons for a sift is $\approx 2\lg n$. This is done for each element in the tree so worst case we get $\approx 2n\lg n$. But heap shrinks upon each iteration (it removes an element) so we take the sum from 0 to $n-1$.

$$\begin{aligned}
\sum_{i=0}^{n-1} 2\lg(i+1) &\approx 2\sum_{i=1}^{n} \lg i \\
&= 2[\lg 1 + \lg 2 + \lg 3 + \cdots + \lg n] \\
&= 2\lg(1 \times 2 \times 3 \times \cdots \times n) \\
&= 2\lg(n!) \\
&= 2\lg\left[\left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n}\right] \\
&\approx 2\left[n\lg\left(\frac{n}{e}\right) + \frac{\lg(2\pi n)}{2}\right] \\
&= 2n\lg n - 2n\lg e + \lg n + \lg(2\pi) \\
&= 2n\lg n + O(n)
\end{aligned}$$

From this we see that even while the tree shrinks, it does not shrink fast enough to make some notable difference. We are still doing $2n\lg n$ comparisons.

7.4 Implementation

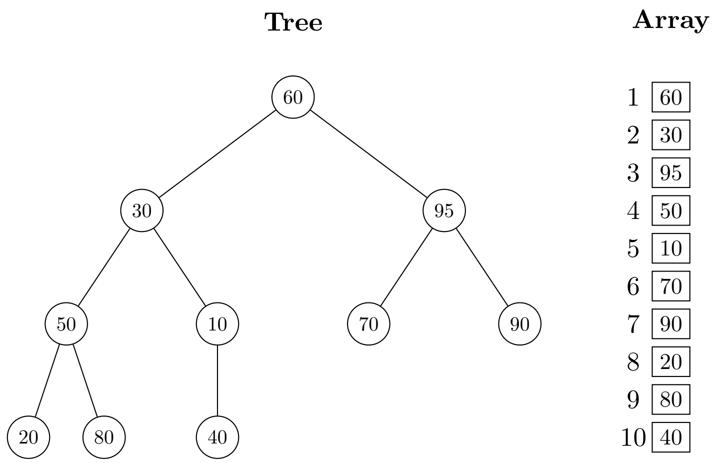


Figure 1: Use an array to implement a tree.
Node has index i, Left child is $2i$, Right child is $2i+1$, and Parent is $\lfloor \frac{i}{2} \rfloor$.

Create

The first parent is at index $\lfloor \frac{n}{2} \rfloor$. Start there and sift down during heap creation. Siblings can be reached by adding or subtracting 1. The result is a created heap.

Finish

To finish the sort we push bottom into tmp first, then move heap root into the bottom most spot.

7.5 Optimization

As Heapsort stands, the result is worse than merge sort. $\Theta(2n \lg n)$ vs $\Theta(n \lg n)$ shows us that much. We compare tmp against both children and this doubles our total number of comparisons. Instead we can sift the hole left by the root down to the bottom in $\lg n$ comparisons. Then put tmp in the hole and sift it back into position. It follows then that we give up $2n$ comparisons on average. Further optimization can be achieved by binary searching up. This gives Heapsort $n \lg n + n \lg(\lg n) = \Theta(n \lg n)$ performance.

8 Integer Arithmetic

8.1 Addition

As you would expect, the time to add two n-digit numbers is proportional to the number of digits, $\Theta(n)$. Since each digit must be sued to compute the sum.

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 1\,2\,3\,4 \\ +\;5\,6\,7\,8 \\ \hline 6\,9\,1\,2 \end{array}$$

We assume that each digital addition takes constant time (with carry also) and label this constant α. The time to add two n-digit numbers is then αn.

8.2 Problem Size

If we were attempting to determine if a number was prime we would divide it by consectutive primes up to the quare root of that number. In Computer Science we want to approach all problems in terms of the problem size. So we should look at both in tmers of the *number of digits* and not the size of the number. So while the prime identification problem is $\Theta(\sqrt{m})$ where m is the number, expressed in binary as 2^n, n being the number of binary digits results in $\Theta(2^{n/2})$. Giving an exponential time algorithm.

8.3 Multiplication

The elementary approach to multiplication runs in $\Theta(n^2)$ time because every digit on top is multiplied by every digit on the bottom. There are $2n(n-1)$ atomic additions in this elementary approach.

$$\begin{array}{r} 1\,2\,3\,4 \\ \times\;5\,6\,7\,8 \\ \hline 9\,8\,7\,2 \\ 8\,6\,3\,8 \\ 7\,4\,0\,4 \\ 6\,1\,7\,0 \\ \hline 7\,0\,0\,6\,6\,5\,2 \end{array}$$

Recursive Multiplication

By memorizing numbers in large bases, say base 100, 2-digit decimal numbers can be treated as 1-digit numbers in base 100. That makes the multiplication easier, because there will be fewer steps. To generalize this for n-digit numbers, treat them as base $10^{n/2}$ and cut each in half. Each has $n/2$ digits, and we know the base.

Then, recurse through this method until we reach a base we have memorized. Then multiply them as two 2-digit numbers, and pull the answer.

Say we want to multiply two 2-digit numbers, ab and cd, we can then do the following.

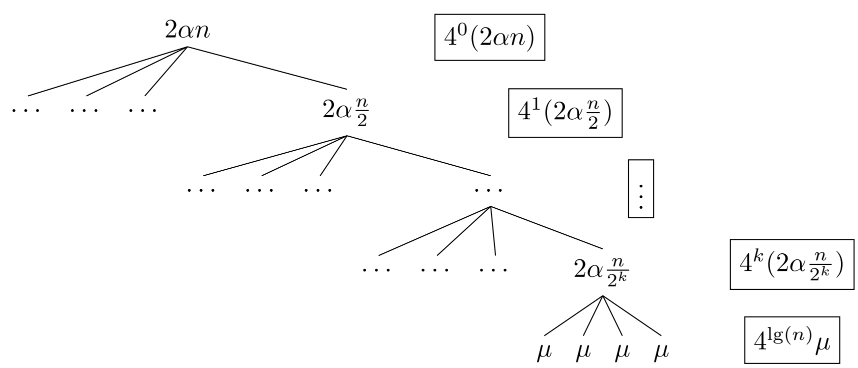
$$ad \times bc = ac + (ad + bc) + bd$$

In the example above ac is an n-digit number as are ad, bc, and bd. When adding, bc is not shifted, bc and ad are shifted by $n/2$ digits, and ac is shifted by n digits. Since ac and bd have no overlap you can add them through concatenation. Then, $ad + bc$ takes αn time and $ac + bd$ is also αn time. This results in a total add time of $2\alpha n$.

So without writing out a formal algorithm,

$$M(n) = 4M\left(\frac{n}{2}\right) + 2\alpha n$$

is the recursion we wish to work on. Now, to solve we assume that the base case time to multiply is μ. The base case is when both of our numbers are 1 digit long, i.e. $M(1) = \mu$.



Now to calculate the total number of atomic actions,

$$
\begin{aligned}
&= \sum_{i=0}^{\lg(n)-1} (4^i(2\alpha\frac{n}{2^i})) + 4^{\lg(n)}\mu \\
&= 2\alpha n \sum_{i=0}^{\lg n-1} \frac{4^i}{2^i} \cdots \\
&= 2\alpha n \sum_{i=0}^{\lg n-1} 2^i \cdots \\
&= 2\alpha n \left(2^{\lg n-1+1} - 1\right) \\
&= 2\alpha n\,(n-1) + 4^{\lg(n)}\mu \\
&= 2\alpha n\,(n-1) + n^{\lg(4)}\mu \\
&= 2\alpha n\,(n-1) + n^2\mu \\
&= 2\alpha n\,(n-1) + n^2\mu
\end{aligned}
$$

So we see that with this method we are doing n^2 multiplications and n^2 additions. In other words, this recursive method has the exact same running time as the elementary method.

Faster Multiplication

9 Quicksort

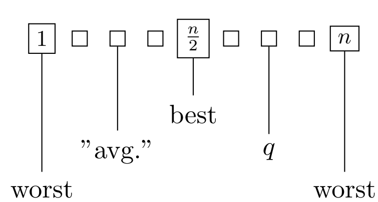
9.1 Pseudocode

Algorithm 8 Quicksort

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
    end if
end procedure

function PARTITION(A, p, r)
    X ← A[r]
    i ← p − 1
    for j = p  to  r − 1 do
        if A[j] ≤ X then
            i ← i + 1
            A[i] ↔ A[j]
        end if
    end for
    A[i + 1] ↔ A[r]
    return (i + 1)
end function
```

9.2 Analysis

The worst case is when the pivot is either in the *first* or *last* index, best case is when the pivot is the *median* index, and the "average" case is when the pivot is between the first/last and median index (here q will represent the *true* average).



Worst Case

As a recurrence,

$$\begin{aligned} T(n) &= T(n-1) + n - 1, \quad T(0) = T(1) = 0 \\ &= \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \end{aligned}$$

Best Case

As a recurrence,

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n - 1, \quad T(0) = T(1) = 0 \\ &= n\lg n - n + 1 \end{aligned}$$

Note: This is the same recurrence as Mergesort!

Average Case

As a recurrence,

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + n - 1, \quad T(0) = T(1) = 0$$

Solve with Strong Constructive Induction.
Guess: $T(n) \leq an\lg n$ for $n \geq 1$ and some constant a.
Base case $n = 1$: $an\lg n = a1\lg 1 = a \cdot 1 \cdot 0 = 0$, $T(1) = 0$ and $0 \leq 0$.
Inductive Hypothesis: Assume true for $< n$, $T(k) \leq ak\lg k$ for $1 \leq k \leq n$.
Inductive Step:

$$\begin{aligned} T(n) &= T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + n - 1 \\ &\leq a\frac{n}{4}\lg\left(\frac{n}{4}\right) + a\frac{3n}{4}\lg\left(\frac{3n}{4}\right) + n - 1 \qquad \text{by IH} \\ &\leq a\frac{n}{4}(\lg n - \lg 4) + a\frac{3n}{4}(\lg 3 + \lg n - \lg 4) + n - 1 \\ &\leq \frac{an}{4}(\lg n - 2) + \frac{3an}{4}(\lg 3 + \lg n - 2) + n - 1 \\ &\leq \frac{an\lg n}{4} - \frac{2an}{4} + \frac{3an\lg 3}{4} + \frac{3an\lg n}{4} - \frac{2 \cdot 3an}{4} + n - 1 \\ &\leq an\lg n + \left(-\frac{a}{2} + \frac{3a\lg 3}{4} - \frac{3}{2}a + 1\right)n - 1 \\ &\leq an\lg n + \left(\left(\frac{3\lg 3}{4} - 2\right)a + 1\right)n - 1 \end{aligned}$$

It follows that we need

$$\left(\frac{3\lg 3}{4} - 2\right) a + 1 \leq 0 \implies a \geq \frac{1}{2 - \frac{3\lg 3}{4}}$$

$$a \gtrsim 1.23$$

Thus,

$$T(n) \lesssim 1.23 n \lg n$$

Exact Average Case

As a recurrence,

$$\begin{aligned}
T(n) &= \left[\sum_{q=1}^{n} \frac{1}{n}\Big(T(q-1) + T(n-q)\Big)\right] + n - 1, \quad T(0) = T(1) = 0 \\
&= \frac{1}{n}\sum_{q=1}^{n}\Big(T(q-1) + T(n-q)\Big) + n - 1 \\
&= \frac{1}{n}\sum_{q=1}^{n} T(q-1) + \frac{1}{n}\sum_{q=1}^{n} T(n-q) + n - 1 \\
&= \frac{1}{n}\sum_{q=0}^{n-1} T(q) + \frac{1}{n}\sum_{q=0}^{n-1} T(q) + n - 1^{*} \\
&= \frac{2}{n}\sum_{q=0}^{n-1} T(q) + n - 1
\end{aligned}$$

*Note that:

$$\sum_{q=1}^{n} T(q-1) = T(0) + T(1) + T(2) + \cdots + T(n-1)$$
$$\sum_{q=1}^{n} T(n-q) = T(n-1) + T(n-2) + T(n-3) + \cdots + T(0)$$

Solve with Strong Constructive Induction.
Guess: $T(n) \leq an\lg n$ for $n \geq 1$ and some constant a.
Base case $n = 1$: $an\lg n = a1\lg 1 = a \cdot 1 \cdot 0 = 0$, $T(1) = 0$ and $0 \leq 0$.
Inductive Hypothesis: Assume true for $< n$, $T(k) \leq ak\lg k$ for $1 \leq k \leq n$.

Inductive Step:

$$\begin{aligned}
T(n) &= n - 1 + \frac{2}{n}\sum_{q=0}^{n-1} T(q) \\
&= n - 1 + \frac{2}{n}\sum_{q=1}^{n-1} T(q) \\
&\leq n - 1 + \frac{2}{n}\sum_{q=1}^{n-1} aq\lg q \qquad \text{by IH} \\
&\leq n - 1 + \frac{2a}{n}\int_1^n x\lg x dx \qquad \text{by integral bound} \\
&= n - 1 + \frac{2a}{n}\left[\frac{x^2\lg x}{2} - \frac{x^2\lg e}{4}\right]\Big|_1^n \\
&= n - 1 + an\lg n - \frac{an\lg e}{2} + \frac{a\lg e}{2n} \\
&= an\lg n + \left[1 - \frac{a\lg e}{2}\right]n - 1 + \frac{a\lg e}{2n}
\end{aligned}$$

It follows that we need

$$1 - \frac{a\lg e}{2} \leq 0 \implies a \geq \frac{2}{\lg e}$$

So set $a = \frac{2}{\lg e} \approx 1.39$, we then need

$$\frac{a\lg e}{2n} - 1 \leq 0 \iff \frac{2\lg e}{(\lg e)2n} - 1 \leq 0 \iff \frac{1}{n} - 1 \leq 0$$

Thus,

$$T(n) \lesssim 1.39n\lg n$$

We can realize a more natural formula,

$$T(n) \leq an\lg n = \frac{2n\lg n}{\lg e} = 2n\ln n$$

Theorem
The expected number of comparisons for Quicksort is $\approx 2n\ln n$.

Blum's Exact Average Case

Let $P(i, j)$ be the probability that the ith smallest and jth smallest elements are compared.

Theorem
The average number of comparisons for quicksort is

$$\sum_{1\le i<j\le n} P(i,j)$$

Only matters the first time an element from $A[i], \cdots, A[j]$ is picked as pivot. The probability that the ith and jth smallest elements are compared during the execution of quicksort is:

$$\frac{2}{j-i+1}$$

Then,

$$\begin{aligned}
\sum_{1\le i<j\le n} P(i,j) &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} P(i,j) = \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= 2\sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{1}{j-i+1} \\
&= 2\sum_{i=1}^{n}\left(\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n-i+1}\right) \\
&= 2\sum_{i=1}^{n}(H_{n-i+1}-1) = 2\sum_{i=1}^{n} H_{n-i+1} - 2\sum_{i=1}^{n} 1 \\
&= 2(H_n + H_{n-1} + \cdots + H_1) - 2\sum_{i=1}^{n} 1 \\
&= 2\sum_{i=1}^{n} H_i - 2n \\
&= 2((n+1)H_n - n) - 2n \\
&= 2(n+1)H_n - 4n \approx 2n\ln n
\end{aligned}$$

9.3 Comments

Quicksort Code

Recall the Quicksort procedure,

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
    end if
end procedure
```

When executing quicksort$(A, p, q-1)$ we must stack up quicksort$(A, q+1, r)$ to be executed later. We need to store the pair of index values $(q+1, r)$.

Stack in Action

The left table represents the pivot being the largest element, and the right table represents the pivot being the smallest element.

Pivot	Stack	Pivot	Stack
$A[n]$	$(n+1, n)$	$A[1]$	$\cancel{(2, n)}$
$A[n-1]$	$(n, n-1)$	$A[2]$	$\cancel{(3, n)}$
$A[n-2]$	$(n-1, n-2)$	$A[3]$	$\cancel{(4, n)}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$A[3]$	$(4, 3)$	$A[n-2]$	$\cancel{(n-1, n)}$
$A[2]$	$(3, 2)$	$A[n-1]$	$\cancel{(n, n)}$
$A[1]$		$A[n]$	

When the pivot is the smallest element the stack remains very small but if the pivot is the largest element we stack $n-1$ problems to do later.

In Place

On average, height of the stack for Quicksort is $\Theta(\log n)$. Height of stack for Mergesort is $\lg n + O(1)$.

Definition: An algorithm is *in place* if it uses $O(1)$ extra variables and (on average) $O(\log n)$ extra index variables.

A more technical definition is; An algorithm is *in place* if it uses at most $O(1)$ extra variables and (on average) $O\left((\log n)^2\right)$ extra bits.

Stack

We can modify the Quicksort procedure so that stack has height at most $\lg n$.

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q ← PARTITION(A, p, r)
        if q ≤ (p + r)/2 then
            QUICKSORT(A, p, q − 1)
            QUICKSORT(A, q + 1, r)
        else
            QUICKSORT(A, q + 1, r)
            QUICKSORT(A, p, q − 1)
        end if
    end if
end procedure
```

Is Quicksort In Place?

Quicksort is in place, but it does not use a constant amount of extra space. Quicksort uses slightly more than a constant amount of extra space.

Choice of Pivots

It is risky to pivot on the last element because the last element could be the largest element. Some better ways could be;

- Pivot on middle element.
- Pivot on median of first, middle, and last elements.
- Pivot on random element.
- Pivot on median of three random elements.
- Pivot on median of five random elements. (Law of diminishing returns.)
- Randomly permute array before starting. (Equivalent to pivoting on random element.)

Partitioning

There are a variety of partition routines. The current edition of the textbook has a version that uses $n-1$ comparisons, but the previous edition of the textbook has a version which uses n comparisons.

A Comparison of Sorting Algorithms

A.1 Temporal and Spatial Locality

Temporal

Temporal locality is dependent on how many variables you have and often you have to replace them.

$sum \leftarrow 0$
$j \leftarrow 1$
for $i = 1$ **to** 10 **do**
  $sum \leftarrow sum + j$
  $j \leftarrow j + j$
end for

The above program has good temporal locality if there are three registers, but bad temporal locality if there are only two registers (there are three unique variables in the program above so anything with less than three avaliable registers will have bad temporal locality).

Spatial

If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. In this case it is common to attempt to guess the size and shape of the area around the current reference for what it is worthwhile to prepare faster access for subsequent reference.

A.2 Comparison Table

Algorithm	Worst Comp.	Avg. Comp.	Worst Moves	Avg. Moves	Worst Exch.	Avg. Exch.	In Place	Spatial Locality
Bubble Sort	$n^2/2$	$n^2/2$			$n^2/2$	$n^2/2$	Yes	Yes
Insertion Sort	$n^2/2$	$n^2/4$	$n^2/2$	$n^2/4$			Yes	Yes
Selection Sort	$n^2/2$	$n^2/2$			n	n	Yes	Yes
Mergesort	$n \lg n$	$n \lg n$	?	?			Y/N	Yes
Heapsort	"$n \lg n$"	$n \lg n$	"$n \lg n$"	$n \lg n$			Yes	No
Quicksort	$n^2/2$	$1.4n \lg n$	?	?			Yes	Yes