# Notes 1/12/2017

*Check that I have **3rd** edition of Algorithms textbook*

The 2 lowest HW scores are dropped.

Exams are on Monday Feb. 6th and Friday March 3.

The website is at http://web.cs.wpi.edu/~gsarkozy/4120/cs4120.html (http://web.cs.wpi.edu/~gsarkozy/4120/cs4120.html)

## 2 Topics: Design and Analysis

- Analysis: (in this class, we mostly care about time complexity)
    - Worst case time complexity (longest running time on any input of a certain size)
    - Average case time complexity (average running time)
        - How this generally works is that we assume that all possible inputs are equally likely, and then we take the expected value (same of mean) of the running time. (Deterministic algorithm with a probabilistic analysis)
    - Randomized algorithms, we use *expected running time*

## Divide and conquer *algorithm design technique*

1. Divide problem into sub-problems
2. Conquer sub-problems by solving them recursively
3. Combine solutions to the sub problems

#### *Merge sort*

Example of a divide and conquer algorithm

Input: An array of N numbers Divide: Two subarrays of size N/2 Conquer: Sort each sub array recursively Combine: Combine the sub problems

```
def Mergesort(A, p, r):
    if p < r:
        q = math.floor((p+r)/2)
        Mergesort(A,p,q)
        Mergesort(A, q+1, r)
        Merge(A, p, q, r)

# Takes at most N comparisons
def Merge(A, p, q, r):
    pass
    # Let's not worry about how this works
```

***Analysis of Divide & Conquer:***

Let's assume our algorithm has input size $n$ and creates $a$ subproblems of (equal, for now) size $n/b, a \geq 1, b > 1$

$$T(n) = \text{(worst case) running time}$$
$$T(n) = a * T(n/b) + dividing(n) + combining(n)$$

Has 3 methods of determining worst case running time:

1. Recursion tree
2. Substitution
3. Master theorem

***For merge sort:***

$$T(1) = C_1$$
$$T(n) = 2 * T(n/2) + C_2 * n$$

Eventual solution is: $T(n) = O(n \log n)$

***Recusing tree method of analysis:***

"Unfold" the recursion: (Assuming n=2^k) C*n* -> Cn/2 + Cn/2 -> C*n/4* + Cn/4 + C*n/4* + Cn/4 -> ... -> C + C + ...

We have $1 + \log n$ levels, and each level takes C*n time (each level sums to* Cn).

$T(n) \leq Cn(1 + \log n) = O(n \log n)$

***Subsitution method of analysis***

1. Guess the form of the solution

2. Verify guess by induction

Basis: $P(1)$ is T (or $P(\text{some other low value})$)

Inductive step: $\forall n(P(1) \wedge \ldots \wedge P(u)) \rightarrow P(n+1) \implies \forall n P(n)$

**Merge sort:**

Assumption: T(n) <= cnlogn

- Basis: T(1) = 1
- T(n) = 2T(n/2) + n
- T(n/2) <= C n/2 log(n/2) (assuming our assumption is right)

```
Basis: n=1:
T(1) = 1
n=2:
T(2) = 4 <= C*2*log2 = 2c
if c >= 2
```

**Incorrect application of the subtitution method (on merge sort):**

Assumption: T(n) = O(n) $\iff$ T(n) <= C n `T(n/2) <= C*n/2`
`T(n) = 2*C*n/2 + n = cn+n = (c+1)n = O(n)` The error is that T(n) <= C n isn't T(n) <= (C+1) *n

# Friday Jan 13

## Master Theorem

T(n) = O(1) if n <= c else: a*T(n/b)+f(n)

### *Main idea of master theorem*

max(f(n), n^(log_b a)

- $O(\ldots) =$ Upper bound
- $\Omega(\ldots) =$ Lower bound
- $\Theta(\ldots) =$ Bounded on both sides

### *Master Theorem*

- Case 1: If $f(n) = O(n^{log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$
- Case 2: If $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(f(n) \cdot \log n)$

- Case 3: If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some $\epsilon > 0$ and we have the regularity condition $a \cdot f(n/b) \leq c \cdot f(n)$ for some $c < 1$ and $n \geq n_0$, then $T(n) = \Theta(f(n))$
- Else, we can't apply the Master Theorem

***Why does this work? (about the proof)***

(10 pages in the book)

$$T(n) = a \cdot T(n/b) + f(n)$$

Recursion Tree: T(n) = a \cdot T(n/b) + f(n) = a(a(T(n/b) + f(n/b))+ f(n) = ...

```
Written as a tree
f(n)
f(n/b) f(n/b) f(n/b)... (a times)
f(n/b^2) ... (a^2 times)
...
1, 1, 1, ... (a^{log_b n} which is eqv. to: n^{log_b a}

Has log_b(n) layers (n = b^k, so the problem has a size of 1)
```

- In Case 1, the bottom level dominates, which is $n^{\log_b a}$
- In Case 3, the top level ($f(n)$) dominates. (regularity condition catches bottom level, polynomially large catches the rest)
- In Case 2, all the levels are roughly the same.

***Ex 1:***

```
T(n) = 9 T(n/3) + n
a=0, b=3, f(n) = n

f(n) = n, also using: n^(log_3(9))) = n^2

Case 1:
   We can pick epsilon=1
   n=O(n^(2-1)) = O(n), which equals f(n), so
   T(n) = Theta(n^2)
```

***Ex 2:***

```
T(n) = 1 T(2n/3) + 1
a = 1, b = 3/2, f(n) = 1
f(n) = 1, also using n^(log_3/2(1))

Is Case 2:
    T(n) = Theta(log n)
```

***Ex 3:***

```
T(n) = 3T(n/4) + nlogn
f(n) = n log n,
also using n^(log_4 (3)) (or, roughly n^0.8

Is Case 3:
    nlogn = Omega(n^(0.8+0.2or_similar))
    Regularity condition: 3/4 • log(n/4) <= 3/4 n log n = 3/4 f(n
), c = 3/4
    T(n) = Theta(n log n)
```

***Ex 4:***

```
T(n) = 2T(n/2) + n log n
f(n) = n log n, n^1

NOT covered by Master Theorem (Case 3 is closest).
( `n log n` isn't polynomially larger than `n^1`)
```

***Ex 5: Merge sort:***

```
T(n) = 2T(n/2) + n
f(n) = n, n^1

Is Case 2:
    T(n) = Theta(n log n)
```

# Tuesday January 17

In the book there are two more divide-and-conquer algorithms:

1. **Maximum sub array problem:** Given an array of $n$ numbers (may be negative), find the subarray that has the maximum sum. (Takes T(n) = 2T(n/2)+O(n), thus Theta(n log n)
2. **Strassen Matrix multiplication algorithm:** T(n) = 7T(n//2)+O(n^2), thus

T(n)=Theta(n^2.7) (roughly)

## Probabilistic analysis & randomized algorithms (Ch. 5)

Random Variables are really a function from the sample space to real numbers.

**Indicator random variable**: A binary (0, or 1) random variable. Traditionally 1 if event occurs, 0 if not.

#### Interview problem

```
for i=1 to n
    interview candidate[i]
    if candidate[i] is better than previous best
        best = i
        hire candidate[i]
```

How many people do we hire?

- Interview cost: $C_i$
- Hiring cost: $C_h$

Cost: $O(C_i n) + O(C_h \cdot m)$ (where $m$ = number of people we actually hire/replace) Worst case $m = n$, if the interviewees come in increasing order.

##### *What is the average hiring cost ($m$)*

The average hiring cost is logarithmic ($O(\log n)$ ), but let's show that:

Assumption: Candidates in random order

$X_i$ = 1 if we hire candidate i, 0 otherwise.

$\Sigma E(X_i) = \Sigma(\text{Probability candidate i is hired})$ Probability candidate i is hired = $1/i$

$\Sigma E(X_i) = \Sigma(1/i)$ = Harmonic sequence ($H_n$) = $\ln(n) + O(1)$

# Thursday January 19th

Probabilistic analysis & randomized algorithms cont.

## Birthday Paradox

K people in a room. When do we expect to have 2 people with the same birthday?

Assumption: Birthdays are uniformly random, 365 days/year.

Worst case: 366 people

Expected:

$k = O(sqrt(n)) \approx 28$

X= # of pairs of people born on the same date.

Goal: $E(X) \geq 1$

$X_{ij}$ `= i and j have same birthday ? 1 : 0;`

$E(X) = E(\Sigma X_{ij}) = \Sigma(E(X_{ij})) = \Sigma(Pr(\text{i and j ave same birthday}) = \text{k choose } 2 \cdot 1/n = k(k -$

$$k(k-1)/(2n) \geq 1$$
$$k(k-1) \geq 2n$$

....

E(1) = 0/365, E(2) = 1/365, E(3) = E(2) + 2/365

E(k) = E(k-1)+(k-1)/365

## Formula satisfiability problem

Note: this is NP complete.

SAT

We have a boolean formula using logical operators.

Is there a satisfying truth assignment?

CNF SAT (Three conjunctive normal form) = simplified transformation of boolean formula. = Conjunction of clauses, each of which are a disjunction of 3 different boolean variables.

#### ***But there's a simple randomized algorithm that works pretty well!***

Expected number of clauses satisfied ($\geq 7/8$)

- Random truth assignment (each variable is true 0.5 of the time)
- X = # of clauses satisfied ($E(X) = 7/8k$)
- $X_i$ = 1 if clause is satisfied
- $\Sigma(Pr($Clause i is satisfied$)$
- Each clause has an expected value of 7/8.

## Heaps and heap sort

- Optimal: $O(n \log n)$
- In place sort
- C uses a combination of this and quicksort

Using a (binary) heap. (Almost full binary tree, depth = O(log n))

[root, child1, child2, child11, child12, child21, child 22...]

```
# NOTE: may not actually run

def parent(i):
    return int(i/2) # int() floors
def left(i):
    return i*2
def right(i):
    return 2*i+1

# Makes sure location i is correct relative to its children
# O(log n)
def maxHeapify(heap, i, heap_size=None):
    if not heap_size:
        heap_size = len(heap)

    left_child = heap[left(i)-1] if not left(i) > heap_size else
float("-inf")
    right_child = heap[right(i)-1] if not right(i) > heap_size el
se float("-inf")
    parent = heap[i-1]
    if parent >= max(left_child, right_child):
        return heap
    elif left_child > right_child:
        heap[left(i)-1] = parent
        heap[i-1] = left_child
        return maxHeapify(heap, left(i))
    else:
        heap[right(i)-1] = parent
        heap[i-1] = right_child
        return maxHeapify(heap, right(i))

# Makes a heap
# O(n)
def buildMaxHeap(llist):
    heap = llist

    for i in range(len(heap)//2, 1, -1):
        heap = maxHeapify(heap, i)
    return heap
# We'd assume this takes O(n log n), but:
# => the depth of the heap being maxHeapified isn't always log(n)
# => At any given height (set of values of i), maxHeapify takes O
(height)
# The number of leaves on any binary tree <= n/2, the number at h
eight 1 is <= n/4.
# At height h, there are <= n/(2^(h+1)) nodes
# This means O(0)*n/2 + O(1)*n/4 + O(3)*n/8...

# O(n log n)
def heapSort(llist):
    # O(n)
    heap = buildMaxHeap(llist)

    for i in range(len(heap), 1, -1): # Do log(n) operation n tim
es
        heap[0], heap[i-1] = heap[i-1], heap[0]
        heap = maxHeapify(heap, 0, heap_size=i)

    return heap
```

Build max heap discussion cont.: Runtime =
$\Sigma_{h=0,h\leq\log(n)}(O(h) * n/2^{h+1}) = O(n) * \Sigma(h/2^{h+1}) = O(n) *$ well known converging series =

## Friday Jan 20

**Priority queue:**

A data structure set S of objects each with a key:

Supports 4 options:

1. Maximum(S)
2. Extract-Max(S)
3. Insert(S, x)

4. Increase-key(S,x,k) (raise priority)

We can use a **max-binary-heap** for this data structure:

```
# This is in **pseudocode**

# O(1)
def maximum(heap):
    return heap[0]

# O(log(n))
def extract_max(heap):
    max = heap[0]
    heap[0] = heap.last # Last may or may not be smallest
    heap.size--
    maxHeapify(heap, 0)
    return max
# O(log(n))
def insert(heap, x):
    heap.size++
    heap.last = x
    increase_key(heap, x, k)
# O(log(n))
def increase_key(heap, x, k):
    # Find x
    # Magic! (don't know how this would be fast)
    # You could use hashmap magic or x could be a reference to a
node of a linked tree.

    x.priority = k
    # Floating up:
    # O(log(n))
    while x.parent && x.parent < x:
       heap[x.pos], heap[x.parent.pos] = heap[x.parent.pos], heap
[x.pos]

# My own idea... not part of priority queue contract
def decrease_key(heap, x, k):
    x.priority = k
    maxHeapify(heap, x)
```

# Monday January 23

## Quicksort:

A[p...r]

1. "Partition around" the pivot element
   - A[p...q-1 < A[q] < A[q+1...r]
2. Recursively sort the two subarrays
3. Combining is trivial.

```
def Quicksort(A, p, r):
    if p < r:
        q = Partition(A, p, r)
        Quicksort(A, p, q-1)
        Quicksort(A,q+1, r)
def Partition(A, p, r):
    # A becomes: [<x..., >x..., undecided..., x]
    # Pivot element is at A[r-1]
    i=0
    for j in range(p, r-1):
        if A[j] < A[r-1]:
            A[i], A[j] = A[j], A[i]
            i+=1
    A[i+1], A[r-1] = A[r-1], A[i+1]
```

### *Analysis*

- Worst case: Every pivot element is a minimum or maximum of the subarray. This can happen if the array is already sorted.
$$T(n) = T(1) + T(n-1) + cn = O(n^2)$$
- Best case: Pivot is centered
$$T(n) = 2T(n/2) + cn = O(n \log n)$$
- Okay case: Pivot is off center.
$$T(n) = T(n/10) + T(9n/10) + cn = O(n \log n)$$
(Use recursion tree method: depth is $log_{10/9}$ width at each level: $cn$

### *Analysis of randomized version:*

Showing $E(X) = O(n \log n)$

- We bound the total # of comparisons $Z_{ij} = Z_i <\dots Z_j$
- When do we compare $Z_i$ and $Z_j$?
- We perform comparisons only when one of the elements is a pivot element.

  - We can *never* compare the same two numbers twice.
- $X_{ij}$ = 1 if $Z_i$ and $Z_j$ are compared.
- $X = \sum_{n-1} \sum_n X_{ij}$
- $E(X) = E(\sum_{n-1} \sum_n X_{ij}) = \sum_{n-1} \sum_n E(X_{ij})$
- Either $Z_i$ or $Z_j$ is a pivot element.
  - In the interval $Z_{ij}$
    - If $Z_i < \text{pivot element} < Z_j$, then they will never be compared.
    - Thus $X_{ij}$ = Probability that $Z_i$ or $Z_j$ is the 1st pivot element from $Z_{ij}$, which is $\frac{2}{\text{interval length}} = \frac{2}{j-i+1}$
- $\sum_{i=1...n-1} \sum_{k=1,n-i} \frac{2}{k+1}$ = $n * \text{Harmonic sequence} = O(n \log n)$

If we can find the median quickly (which we can do in $O(n)$), then we can make the worst case $O(n \log n)$

## Sorting cannot be improved beyond $O(n \log n)$

For any comparison-based sorting algorithm, it must take $\geq n \log n$ steps in the worst case. $\Omega(n \log n)$

1. Decision tree method: (It's very rare that the decision tree method will give the lowest bound)
2. Adversary argument: (much better) "Adversary" trys to make your algorithm look as bad as possible.

#### Binary search tree for comparison based sorting

Sorting algorithms are like Binary search trees. Each comparison gives us two options, we can build each of these possibilities as a binary tree, where leaves are when the algorithm finishes, which should be contain all the $n!$ unique permutations (or have repeats - but that's inefficient.)

So, we have at least $n!$ leaves. Each execution of the algorithm follows a path from the root to a leaf.

$$n! \leq \text{\# of leaves} \leq 2^{\text{height of tree}} = 2^{\text{worst case running time}}$$

$$n! \leq 2^{\text{worst case running time}}$$

$$\log_2 n! \leq \text{worst case running time}$$

It's fairly obvious that $n! = n * (n-1) * \ldots (n/2) * (n/2 - 1) \ldots 2 * 1 > (n/2)^{n/2}$, thus:

$$\log_2 (n/2)^{n/2} < \log_2 n! \leq \text{worst case running time}$$
$$n/2 * \log_2 (n/2) < \text{worst case running time}$$
$$\Omega(n \log_2 n) = \text{worst case running time}$$

A tree with $n!$ leaves *must* have height of $n\log_2 n$ or more.

Merge sort and heap sort are optimal (at least as far as $O$ notation goes)

#### ***General equation for (binary) search tree problem:***

$$X \leq \text{\# of possible answers} \leq 2^h$$
$$h \geq logX$$
$$h = \text{worst case running time, also the tree height}$$

This is usually a pretty weak lower bound.

## Exceptions - non-comparison based sorting algorithm with $O(n)$

#### ***Counting sort***

Assume that all numbers come from the range $0, 1, \ldots k$ where $k = O(n)$ Algorithm runs in $O(n + k)$.

Doing this *stable* (equal elements are left in the same relative position).

```
def count_sort(input, max):
    C = [0]*max # Counts or indexes
    output = [-1]*len(input)

    # Get the counts
    for i in range(len(input)):
        C[input[i]] += 1

    # Sum the counts
    # (any item with index i will be at or before output[C[i]])
    for i in range(1, max):
        C[i] = C[i] + C[i-1]

    # Put elements into the output array
    # Start from back to make it stable
    for i in range(len(input), -1, -1): # i = n downto 0
        output[C[input[i]]] = input[i]
        C[input[i]] += -1
    return output
```

## Selection problem: Find the ith smallest element (or other statistic)

Special cases:

- $i = 1$ minimum
- $i = n$ maximum
- $i = \frac{n+1}{/}2$ median. Note: if $n$ is even, we probably want to find two medians.

Can be done in $O(n)$!

Finding the **Maximum** is "exactly typed" with $n - 1$ comparisons.

#### *(simple) Adversary argument for finding the maximum*

Each player who is *not* the max must be compared and be smaller than at least one element, thus at least $n - 1$ matches (e.g. comparing the max against each element, or something else similar).

We call the way of thinking about matches this way as the **Tournament method**

Suppose that there is an algorithm such that there are two elements $x$ and $y$ who never lost.

Our algorithm must declare one of them (say $x$) as the winner.

But if our adversary picks an input such that $y$ is the maximum, then the algorithm would be wrong!

Thus each non-maximum should loose a comparison and there should be at least $n - 1$ comparisons.

## Simultaneous minimum and maximum

Can obviously do this in $2(n - 1)$, but we can do better.

We can do it in $3/2 * (n - 1)$

We compare pairs to each other, then compare the larger of the two to max, the smaller to min.

- For $n$ is odd, we have $3 \lfloor n/2 \rfloor$ (start with A[0] as both max and min)
- For $n$ is even, we have $3n/2 - 2$ (start with max(A[0], A[1]) and min(A[0], A[1])

```
def min_max(A):
    min = A[0]
    max = A[0]
    for i in range(1, len(A)/2):
        A[i*2-1] =
        A[i*2] =
```

**Proof that this is the best running time**

Let's create 4 sets: $(B, W, L, O)$

- $B$ - beginners
- $W$ - winners, no losses, some wins
- $L$ - losers, some losses, no wins
- $O$ - others, some wins and losses

Initially: $(n, 0, 0, 0)$

At the end: $(0, 1, 1, n-2)$

***Adversary's goals:***

1. Nothing should go directly from $B$ to $O$. (this slows things down)
2. From $W$ and $L$, they go to $O$ one by one.

Suppose the Adversary achieves this goal.

We need at least $\lceil n/2 \rceil$ comparisons to empty $B$ We need $n-2$ comparisons to empty $W$ and $L$.

| **x:y** | $(B, W, L, O)$ |
|---|---|
| B:B | $(B-2, W+1, L+1, O)$ |
| B:W | $(B-1, W, L+1, O)$ |
| | $(B-1, W, L, O+1)$ - adversary eliminates |
| B:L | $(B-1, W+1, L, O)$ |
| | $(B-1, W, L, O+1)$ - adversary eliminates |
| B:O | $(B-1, W+1, L, O)$ |
| | $(B-1, W, L+1, O)$ |
| W:W | $(B, W-1, L, O+1)$ |
| W:L | $(B, W, L, O)$ |
| | $(B, W-1, L-1, O+2)$ - adversary eliminates |
| W:O | $(B, W, L, O)$ - adversary prefers |
| | $(B, W-1, L, O+1)$ |
| L:L | $(B, W, L-1, O+1)$ |

| | |
|---|---|
| L:O | $(B, W, L, O)$ -adversary prefers |
| | $(B, W, L-1, O+1)$ |
| O:O | $(B, W, L, O)$ |

The adversary can control the $B$ set, so it can control any comparison with it, but it is less obvious how it can control the comparisons with just the other 3.

```
# A=Array
# p=partition start
# r=partition end
# i=number of elements smaller than the result (e.g. 0 for min(A)
)
def Randomizedselect(A, p, r, i):
    q = RandomizedPartition(A, p, r)
    k = q-p+1 # Rank
    if i==k:
        return A[q]
    elif i < k:
        return RandomizedSelect(A, p, q-1, i)
    else:
        return RandomizedSelect(A, q+1, r, i-k)
```

Has an expected running time of $O(n)$, but a worst case running time of $O(n^2)$

1. Divide into groups of 5 elements
   - Why 5? It's odd, so we can find the median.
2. Find the median of each group $O(n) * O(5)$
3. Find the median of the medians ($x$) by repeating step 2
   - $O(n/5)$
4. Partition around this median of medians $k$ = rank of $x$
5. if $i = k$, return $x$ if $i < k$, Recursively find ith on the low side if $i > k$, recursively find $(i-k)^{th}$ on the high side

This guarantees a fairly even split and thus $O(n)$ running time. The split is fairly even because $x$ cannot be in the highest or lowest quartile. $x$ is lower than half of the medians, and thus is lower than 3/5 of their elements. $x$ is smaller than $3(0.5 * \frac{n}{5} - 2) = 0.3n - 6$ (-2 is beccause the 1st group may not contain all 5 elements.)

$$T(n) \leq T(0.7n + 6) + T(n/5) + an$$

We can use the **substitution method** to find $T(n) \leq cn$ if $c$ is large enough compared to $a$

Assume $T(n) \leq cn$.

$$T(n) = c \cdot 0.7n + 6 + c \cdot n/5 + an \leq cn$$
$$\frac{9cn}{10} + an + 6c \leq c \cdot n$$
$$an + 6c \leq \frac{c}{10}n$$

## January 30

## Binary Search Trees

- Can perform an `in-order-tree-walk` to get a sorted list

```
def inOrderTreeWalk(root):
  if root == None:
      return []
  return inOrderTreeWalk(root.left) + [root.key] + inOrder
TreeWalk(root.right)
```

- Finding an item in the tree is $O(\text{height})$

```
def TreeSearch(key, root):
    if root == None:
        return None # key is not in the tree
    if root.key == key:
        return root
    elif root.key > key:
        return TreeSearch(key, root.right)
    else:
        return TreeSearch(key, root.left)

def TreeInsert(key, root):
    child = None
    if root.key > key:
        child = root.right
    else:
        child = root.left
    if child != None:
        TreeInsert(key, child)
    else:
        if root.key > key:
            root.right = Node(key, parent=root)
        else:
            root.left = Node(key, parent=root)

def TreeMinimum(root):
    while root.left != None:
        root = root.left
    return root.key
```

Removing from a tree is a bit trickier.

```
def TreeSuccessor(root):
    if root.right == None:
        # Go up on the left until the 1st right turn
        y = root.parent
        while y != None and root == y.right:
            root = y
            y = y.parent
        return y
    else:
        return TreeMinimum(root.right)

def TreeRemove(root):
    if root.left == None:
        root.parent.appropriateChild = root.right
        if root.right != None:
            root.right.parent = root.parent
    elif root.right == None:
        root.parent.appropriateChild = root.left
        root.left.parent = root.parent
    else: # We have 2 children
        successor = TreeSuccessor(root) # Successor has at most o
ne child because root has two children
        TreeRemove(successor) # This runs in O(1) time because su
ccessor has no left child
        #successor.parent.appropriateChild = successor.right
        #if successor.right: successor.right.parent = successor.p
arent
        root.parent.appropriateChild = successor
        successor.parent = root.parent
        successor.left = root.left
        successor.right = root.right
```

The height is not necessarily $O(log(n))$ - e.g. if the keys come in sorted order.

If the keys are inserted (with no deletions) in random order, $E(\text{height}) = O(\log(n))$.

This takes a while to show, but seems straightforward, it proves easiest to use the *exponential height* $Y_n = 2^{X_n}$ and $Z_{n,i}$ `=1 if the rank of the root is i.` $E(Z_{n,i}) = 1/n$

## Greedy algorithms

### Activity selection problem

Set of activities $a_1, a_2 \ldots$ are competing for a resource for each activity $a_i$:

- $s_i$ = starting time
- $f_i$ = finishing time
- $a_i = [s_i, f_i)$

Activities $a_i$ and $a_j$ are compatible if their times don't overlap.

**Goal**: Find a maximum set of mutually compatible set of activities.

1. Sort activities by finishing time! ($O(n \log n)$)
2. Greedy choice: We select 1st the activity that finishes 1st ($a_1$)
3. remove activities that start before $f_1$
4. repeat as necessary

***Claim for greedy choice to work:***

Let $a_m$ be the activity that finishes 1st in $S_k$ $a_m$ is part of an optimal solution for $S_k$

Let $A_k$ be an optimal solution for $S_k$.

- If $a_m \in A_k$ - our assumption is fine
- Else $a_m \notin A_k$:
    - We can replace $a_j$ with $a_m$ where $a_j$ is the element in $A_k$ that finishes first.
        - $a_m$ *must* finish before $a_j$, so $A_k - a_j + a_m$ is still compatible.
        - $A_k - a_j + a_m$ is just as optimal as $A_k$.

Thus the greedy choice provides a globally optimal solution.

```
# Actually psuedocode

activities.sort_by("finishing_time") # O(nlogn)

finishing_time = 0
while not activities.empty(): # O(n^2)
    finishing_time = activities.dequeue().finishing_time
    activities = activities.filter(lambda x: x.starting_time > fi
nishing_time)
```

Or, more efficiently:

```
activities.sort_by("finishing_time") # O(nlogn) - though we can p
robably do a linear time sort!

finishing_time = 0
while not activities.empty(): # O(n)
    a = activities.dequeue()
    if a.starting_time > finishing_time:
        finishing_time = a.finishing_time
```

## Amortized analysis

Let's say we have a sequence of $n$ operations that can have very different running times.

We want to average over $n$ operations.

This is *very* different from average case analysis. (Amortized = average over operations, no probability. Average-case = average over all inputs)

#### *Multipop stack example*

3 operations:

- Push(S, x) $O(1)$
- Pop(S) $O(1)$
- Multipop(S, k) (pops $min(|S|, k)$ elements. $O(k) \leq O(n-1) = O(n)$ or $O(1)$ depending on implementation

#### *Binary counter example*

Have $k$ bits, cost is number of bits we have to flip.

We can consider the counter equal to $x = \sum_{i=0...k-1} A[i] * 2^i$

So, if $k = 4$,

- 0 -> 1: 1 bit flipped
- 1 -> 2: 2 bits flipped
- etc. upt until 15 -> 0: 4 bits flipped.

Worst case for one operation $O(nk)$.

#### Aggregate method

$$T(n) = \text{total bound for the cost of the n operations}$$

$$\text{amortized cost (aggregate method)} \leq \frac{T(n)}{n}$$

***Multipop stack example***

If we have $n$ operations, which might be multipop, $T(n) = O(n * n) = O(n^2)$. But we can do better.

Let's look at the total number of pops: we can have $n - 1$ pops ($\leq$ number of pushes $\leq n$)

$$T(n) \leq 2n = O(n)$$

$$\text{amortized cost } \leq \frac{T(n)}{n} = \frac{O(n)}{n} = O(1)$$

***Binary counter example***

We have to flip the last bit $n$ times, 2nd bit $n/2$, 3rd $n/4$... etc. So the total flips is:

$$T(n) \leq \sum_{i=0...k-1} \frac{n}{2^i} \leq n \sum_{i=0...\infty} = 2n$$

$$\text{amortized cost } \leq \frac{T(n)}{n} = \frac{O(2n)}{n} = O(2)$$

**Accounting method**

- $c_i$ = actual cost of operation i.
- $\bar{c_i}$ = amortized cost.
  - If we guess too high, we can consider the extra as credit for later operations. $\sum \bar{c_i} - \sum c_i = \text{credit} \geq 0$
  - We *need* to make sure $\text{credit} \geq 0$

We have different amortized costs for different operations.

***Multipop stack example***

If we overcharge for the push (pretend it costs $O(2)$), we can pop freely (assuming we don't underflow our stack).

Now we need to show $\text{credit} \geq 0$. We put 1 operation in credit for each element in the stack, and the only operations we use to remove elements cost as much as the number of elements they remove, so because # $\text{of plates} \geq 0$, $\text{credit} \geq 0$. We have to assume that we never underflow the stack, however.

Amortized Cost:

- Push: $2$

- Pop: $0$
- Multipop: $0$

***Binary counter example***

We can charge each operation for the number of 1s in the counter.

**Showing** $\sum \bar{c}_i - \sum c_i = \text{credit} \geq 0$:

$\sum \bar{c}_i - \sum c_i$ is the number of ones, so $\sum c_i \leq \sum \bar{c}_i \leq 2n$

#### Potential method

We have a state associated with each possible configuration, and we need a function $\Phi$ that gives us the potential of the state such that:

$$\bar{c}_i = c_i + \Phi(\text{state}_i) - \Phi(\text{state}_{i-1})$$

***Multipop example***

$\Phi$ = # of elements in the stack

- Push $\bar{c}_i = 1 + 1 = 2$
- Pop $\bar{c}_i = 1 + (-1) = 0$
- Multipop $\bar{c}_i = k + (-k) = 0$

***Binary counter example***

$\Phi$ = # of 1s in the counter, which gives us the same result as the Accounting method.

# Dynamic Programming

- Pay attention to subproblems
- Use a table to avoid recomputation
- The subproblems make a `directed acyclic graph`

## Longest Common Subsequence

```
def is_subseq(subseq, y):
    i=0
    for letter in subseq:
        while letter != y[i]:
            i = i+1
            if len(y) <= i:
                return False
        return True

# Brute force method:
import itertools
def dumb_lcs(x, y):
    max_subseq = ""
    # for each subsequence of x
    for length in range(len(x)):
        for subseq in itertools.combinations(iterable, r):
            if len(max_subseq) < len(subseq) and is_subseq(subseq
, y):
                max_subseq = subseq

# Naive, but reasonable method
def naive_lcs(x,y):
    if (not x) or (not y):
        return ""
    else:
        if x[0] == y[0]:
            return x[0] + naive_lcs(x[1:], y[1:])
        else:
            tmp1 = naive_lcs(x, y[1:])
            tmp2 = naive_lcs(x[1:], y)
            return tmp1 if len(tmp1) > len(tmp2) else tmp2
import numpy as np
def lcs(x,y):
    #make_lcs_len_subseq
    table = np.zeros((len(x)+1,len(y)+1))
    for i in range(len(x)):
        for j in range(len(y)):
            if x[i] == y[j]:
                table[i+1, j+1] = table[i,j] + 1
            else:
                table[i+1, j+1] = max(table[i,j+1], table[i+1, j]
)
    # trace back lcs subseq
    subseq = ""
    i = len(x)
    j = len(y)
    while table[i,j] != 0:
        if x[i-1] == y[j-1]: #table[i,j] == table[i-1, j-1]
```

```
                subseq = x[i-1] + subseq
                i += -1
                j += -1
            elif table[i-1, j] == table[i,j]:
                i += -1
            elif table[i, j-1] == table[i,j]:
                j += -1
            else:
                # shouldn't get here
                print('error')
        return subseq

lcs("spanking", "amputation") == "pain" and lcs('abcuvxy', 'bcwxy
z') == 'bcxy'
```

## Graphs: Minimum Spanning tree:

Have a graph $G = (V, E)$, connected, undirected, and weighted with $w(e)$.

$T$ spanning tree where $w(T) = \sum_{e \in T} w(e)$.

**Goal:** Find a MST $T$ where $w(T)$ is minimum.

#### Generic greedy algorithm: (both Kruskal's and Prim's algorithms)

$$O(E \cdot \log V)$$

We'll need to define a few terms. A *Cut* is a split of the graph's vertexes into two groups, and a crossing edge is one that goes from one of the groups or another. The cut $C = (S, V - S)$ *respects* $A$ (a set of edges) if none of the edges of $A$ is a crossing edge.

```
# Pseudocode

A = {} # Set of edges. ALWAYS a part of some spanning tree
while A is not a MST: # len(A) + 1 = len(vertices)
    C = select_cut(respecting=A)
    A += C.crossing_edges().lightest()

def select_cut_prims(respecting={}):
    if respecting == {}:
        respecting = {vertices[0]}
    q = minQueue(vertices[0].adjacent, distance_from_A)
    u=extract_min(q)
    a.add(u)
    q.add(u.adjacent)


def select_cut_kruskals(respecting={}):
    # Graph is given as a neighborhood list
    if !sorted_e:
        sorted_e = edges.sort_by('weight') # This is the majority
of the work O(ElogE) = O(ElogV)

    connected_components = vertices.map(lambda v: [v]) # don't us
e

    while connected_components.any(contains_both_ends(sorted_e[0]
)):
        sorted_e = sorted_e[1:]

    connected_components.filter(contains_an_end(sorted_e[0])).com
bine()
    sorted_e = sorted_e[1:]


# Actual reasonable version of the algorithm:
def primsMST(vertices): # Not quite prim's - Prim's keeps a min q
ueue of vertices+weights, not edges
    mst = []
    verts = [vertices[0]]
    q = minQueue(vertices[0].edges, sort_by_weight)
    while len(verts) != len(vertices):
        u = q.minPop()
        if u.end_two not in verts:
            mst = mst + [u]
            verts = verts + [u.end_two]
            q.add(u.end_two.edges)

def primsMST(vertices):
    mst = []
    verts = [vertices[0]]
    q = minQueue(vertices, weights=+infinity)
    while len(verts) != len(vertices):
        u = q.minPop() # Runs V times, so O(V log V)
        for edge in u.adjacent:
            q.decreaseKey(edge.vertex, edge.weight) # Runs E time
s, so O(E logV)
```

***Show that*** $A_{t=t+1} \in \text{some spanning tree}$ ***given*** $A_{t=0}$

Let us have a spanning tree $T$ that $A$ is part of.

Now we want to show that $A + (u, v)$ is part of some MST.

We know that there is an edge $(x, y) \in T$ where $(x, y)$ crosses any cut (that has non-empty sides).

Let $T' = T - (x, y) + (u, v)$. Then $w(T') = w(T)$

....

$A$ is a forest (but not yet a spanning tree). Let's take one of the connected components ($C$). Let our cut be one of the connected components, and the rest of the graph. The lightest edge of this cut, $(u, v)$ is safe to add to $A$.

---

## Shortest path (directed graph)

$$\text{Given Graph } G\text{, connected weighted, directed graph (as neighborhood list)}$$
We want to minimize:

$$w(\text{path}) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$
$$\delta(u, v) = \min_P w(P)$$

Goal: Find $\delta(s, v) \mid v \in V$

There may be some negative weight edges, but no negative weight cycles in G. (this means there's no cycles in the shortest path, and thus `len(shortest_path) <= len(vertices)-1`.

Also, the sub-path of any shortest path is the shortest path between those vertices.

### Bellman-Ford algorithm

Slightly slower than Dijsktra, but detects negative weight cycles (and allows negative weights)

**is a relaxation method**

$v.d$ = current upper bound on $\delta(s, v)$, it starts at infinity.

```
# Pseudocode assumes:
# w(vertex1, vertex2) = weight function
# edges = [{ u: start vertex, v: end vertex },...]

def Relax(u,v):
    if u.d + (w(u,v)) < v.d: # if distance to u + weight(edge bet
ween u &v)) < distance to v
        v.d = u.d + w(u,v)
        v.predecessor = u # often v.pi= predecessor

def BellmanFord(edges, start, end):

    # relax every edge V-1 times
    for i in range(len(vertices)-1):
        for (u,v) in edges:
            Relax(u, v)

    # check for negative weight cycles
    for (u,v) in edges:
        if u.d + w(u,v) < v.d:
            throw "Negative weight cycle detected

    # Create path
    path = []
    while path[0] != start:
        path = [path[0].predecessor] + path
    return path
```

### *Correctness of BF*

**Path relaxation Property**: If $P = (v_0, v_1, \dots v_k)$ is a shortest path from $v_0 = s$ to $v_k = v$, and we relax the edges $(v_0, v_1), (v_1, v_2), (v_2, v_3), \dots (v_{k-1}, v_k)$ in this order, then at the end $v.d = \delta(s, v)$. This still holds if we perform other relaxations intermixed.

### *Detecting negative weight cycles*

If there is a negative weight cycle, we should be able to detect it:

$$\exists \text{ a cycle } C(v_0, v_1, \dots v_{k-1}, v_0) \mid \sum_{i=1}^{k} w(v_i, v_{i-1})$$

If there is a negative weight cycle, then:

$$\sum v_i.d \le \sum v_{i-1}.d + \sum w(v_{i-1}, v_i)$$

## Dijkstra

Can think of Dijkstra's algorithm as like Prim's algorithm.

We maintain a set $S$ for which we know the shortest path, and we maintain a priority queue of vertices to relax.

Takes $O(E \log V)$, so $O(n^2 \log n)$

```
def dijkstra():
    q = PriorityQueue(start)
    S = {start} # Set of vertices we know the shortest path of
    while not q.empty():
        u = q.extract_min()
        S += u
        for vertex in u.neighbors:
            Relax(u,v) # relax also has to do q.reduce_key(v, ...
) and v.d = ...
```

### *Correctness of Dijkstra*

Note: We cannot use the Path Relaxation Property, because we may relax the edges in the wrong order (with 0 weight edges?)

What we really want to show is that when we add an element to $S$, we must ensure that it has the shortest path.

**Proof by contradiction** Lets assume the opposite:

- Take the first $u \in S \mid u.d \neq \delta(\text{start}, u)$.
- Take a shortest path to $u$ before we add it to $S$
- Along this path, take the 1st vertex $y \notin S$ (it's possible $y = u$)
- $x = y.predecessor$
  - note that $x.d$ is the shortest distance to $x$
  - And we relaxed $(x, y)$, so $y.d = \delta(\text{start}, y)$ must be correct too.
    - $y.d \le \delta(\text{start}, u) \le u.d$ (because no negative edges)
  - Thus y should be in S
  - If $y$ is in $S$, then, $u$ must equal $y$ and $\delta(\text{start}, u) = u.d$

# All pairs shortest path

Find the shortest paths between *all* vertices.

- Naive solution: n-times BF: $O(n^4)$
- Naive solution: n-times Dijkstra: $O(n^3 \log n)$
- Matrix multiplication algorithm $O(n^3 \log n)$ - can handle negatives
- Floyd-Warshall: $O(n^3)$ - still can handle negatives

Both of the new techniques use dynamic programming.

### Matrix multiplication algorithm

We use the Adjacency matrix representation $W = (w_{i,j}) =$ `0 if i=j, w(i,j) elif i and j adjacent, else +infinity`

1. $l_{ij}^{(m)}$ = shortest distance between $i$ and $j$ using $\leq m$ edges.
    - $L^{(m)} = (l_{ij}^{(m)})$
    - $L^1$ = adjacency matrix
    - Goal is to find $L^{n-1}$
2. $L^{m+1} =$
    - Each element of the new L is the minimum of $W_j + L_i$
    $$l_{ij}^m = \min_{1 \leq k \leq n} (l_{ik}^{m-1} + w_{kj})$$
    - Sometimes we write this as $L^m = L^{m-1} \bigodot W$ where $\bigodot$ is a new invention

It's also reasonably easy to maintain a predecesor matrix alongside this one.

```
# Naive solution
for i in range(1,n):
    for j in range(1, n):
        l[i,j] = infinity
        for k in range(1, n):
            l[i,j] = min(l[i,j], l[i,k]+w[k,j]) # Note: if we pla
ced minimum -> +, and + with *, we get matrix multiplication
```

We can improve this via repeating squaring:

- $L^{(1)} = W$
- $L^{(2)} = W^2$ - note: $W^2 = W \bigodot W$
- $L^{(4)} = W^4$
- $L^{2^{\lceil \log(n-1) \rceil}}$

### Floyd-Warshall alrogrithm

$d_{ij}^{(k)}$ = shortest distance between i and j using intermmediate vertices in $\{1, 2, \dots k\}$

- Start with: $D^{(0)} = W$
- Goal: $D^{(n)}$

We're adding one vertex at a time to our set of vertices, so:

$$d_{ij}^{(k)} = min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

```
for k in range(1, n):
    for i in range(1, n):
        for j in range(1, n):
            d[i,j] = min(old_d[i,j], old_d[i,k]+old_d[k,j])
```

Can find **the transitive closure**. Let $G = (V, E)$ be a directed un-weighted graph.

$G^* = (V, E^*)$ where $E^* = \{(i, j) | \exists \text{ path } i \to j\}$

We can find $E^*$ by running Floyd-Warshall and if d[i,j] is finite, then there is a path between i and j.

## Flow networks: Ford Fulkerson algorithm

We start with a directed, connected graph, and we have two special vertices:

- $s$ = source
- $t$ = sink

Think of a flow network as a network of pipes each with a capacity of $c(u, v) = \text{ capacity of edge } (u, v)$ where the source produced 'water' and the sink consumes it. ( $c(u, v)$ where $(u, v) \notin G$ is $0$)

- We must also assume that there are no antiparallel edges (cycles with only two vertices).

Flow =$f(u, v). f : V * V \to \text{Reals}^{\geq 0}$

$$|f| = \text{outflow - inflow to the source (usually 0)}$$
$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Goal: Maximize $|f|$

Antiparallel edges can be fixed by changing 1 -> 2 and 2-> 1 to 1 -> 2, 2 -> new3, new3 -> 1

***Residual network:***

Represents how much we can increase/decrease our flow between two nodes (edges represent amount of 'water' we can fit into the pipes or how much 'water' we can 'remove'. Edges with flow/weight 0 are deleted.

$$G_f = (V, E_f)$$
$$E_f = \{(u,v) | c_f(u,v) > 0\}$$

- $c_f(u,v) = c(u,v) - f(u,v)$ if $(u,v) \in E(G)$
- $c_f(u,v) = f(u,v)$ if $(v,u) \in E(G)$
- otherwise 0

There should be almost twice as many edges in $E_f$ as there are in $E$.

***Augmenting path***

Just a simple path $P$ from the source to the sink in $G_f$

Note that $c_f(P) = \min\{c_f(u,v) | (u,v) \in P\}$ You'll need to augment the flow with $f_p(u,v) = (u,v) \in P ? c_f(P) : 0$ (I used a ternary operation...)

Augmented flow:
$$(f \uparrow f_P)(u,v) = f(u,v) + f_p(u,v) - f_p(v,u) \text{ iff } (u,v) \in E(G)$$
Total augmented flow is $|f \uparrow f_p| = |f| + |f_p|$

***Ford Fulkerson algorithm***

- Initialize $f$ to 0 for all edges.
- while there is an augmenting path $P$ in the residual network $G_f$ (Edmonds-Karp always selects the path with the fewest edges, but that isn't too important)
  - augment $f$ along the path $P$
- Return $f$

**How do we know that the flow is maximum?**

We're showing that if $\nexists$ augmenting path, flow is maximum.

Let a cut be one of $\{(S,T) | S \subset E(G), T \subset E(G), s \in S, t \in T\}$ where $s$ is the source and $t$ is the sink.

Capacity of the cut:
$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

Note that the flow is at most the copacity ofthe cut. $|f| = f(S,T) \leq c(S,T)$ (note that the flow across the cut, $f(S,T)$ is $\sum \sum f(u,v) - \sum \sum f(v,u)$

***Max flow min cut thm: (will continue tomorrow)***

The following are equivalent:

1. $f$ is a maximum flow
2. $\nexists$ augmenting path in $G_f$
3. $\exists$ a cut $(S, T), |f| = c(S, T)$

## Thursday Feb 23

### Max-Flow min-cut theorem (Ford-Fulkerson)

1. $f$ is a max flow
2. $\nexists$ augmenting path
3. $\exists$ a cut $(S, T)$ where $|f| = c(S, T)$

These are all equivalent:

- 1 -> 2 is trivial
- 3 -> 1 is trivial ($|f| = f(S, T) \leq c(S, T)$)
- 2 -> 3 (slightly trickier)

Let us assume there are no augmenting paths in $G_f$.

Let $S = \{u \mid \exists \text{ a path from s to u in } G_f\}$

We have to show that for this particular cut, $|f| = c(S, T)$.

Remember:

$$|f| = f(S, T) = \sum_{u \in S} \sum_{v \in t} (f(u, v) - f(v, u))$$

Because $S$ contains all vertices reachable from $s$, we know that any edges must be filled to capacity thus $f(u, v) = c(u, v)$ and $f(v, u) = 0$, thus $|f| = f(S, T) = c(S, T)$

***Time*** $O(I \cdot E)$

$I$ = # of iterations

Assumptions: Integrality condition: All $c(u, v)$ are integers

In each iteration $|f|$ must increase by at least 1. So, $I \leq f^*$ (maximum flow).

If we don't assume that all $c(u, v)$ are integers, it may be divergent ?

***Edmonds-Karp method***

We always pick the shortest augmenting path (counting edges). $I \leq V/2 \cdot E$

Thus, $O(E^2 V) = O(n^5)$

$(u, v)$ is a critical edge if $c_f(u, v) = c_f(P)$. Thus each edge can become critical at most $\leq V/2$ times.

Let $\delta_f(s, v)$ = shortest distance to v from s in $G_f$ We know that $(u, v)$ is on the shortest path (on $G_f$, not necessarily on $G$) when it becomes critical for the $1^{st}$ time. Thus $\delta_f(s, v) = \delta_f(s, u) + 1$

It can become critical again when we send flow along $(v, u)$, and then reverse it along $(u, v)$ but now, $\delta_f(s, u) = \delta_f(s, v) + 1 = \delta_{f\,\text{old}}(s, u) + 2$

**Application of flow networks for bipartite graph matching**

- Assume we have a: Unweighted bipartite graph G
  - Bipartite graph = graph with two sets of vertices ($L$ and $R$) where vertices in $L$ are only neighbors of vertices in $R$ and vice versa.
  - Matching $M$ = subset of $G$ where all edges are disjoint (don't share edges)

Let's connect $s$ to all the elements in $L$, and $t$ have an edge to all elements of $R$.

The size of the matching (|$M$|) is $|f|$ in this new graph. (note $c(u, v) = 1$ for all edges)

Running time is $O(VE) = O(n^3)$

# P = NP

### $P$ = problems solvable in polynomial time ($O(n^c)$)

To get $P$ algorithm from an Optimization problem, we convert it to a Decision problem.

*Example:* Find a maximum Clique (complete subgraph) of a graph

We take what initially appears to an $NP$ complete problem,

CLIQUE = {(G, k) | G has a clique of size k}

### $NP$ = problems we can verify in polynomial time $O(n^c)$

Another definition is that: Languages accepted by polynomial time Non-deterministic Turing Machine. (NP stands for Non-polynomial).

*Example: Hamiltonian Cycle (unweighted travelling salesman problem)*

Language $L \in NP$ if there is a polynomial time algorithm $A(x, y)$ such that:

$x \in L \iff \exists$ a certificate $y, |y| = O(|x|^c)$ then $A(x, y)$ is yes.

Obviously, $P \subseteq NP$.

**NP Complete** (NPC) are the problems we thing are in $NP$ but not $P$

## Polynomial time reduction

$L_1 \leq_p L_2$ is reducible to $L_2$ in polynomial time if there is a polynomial time computable function $f$ such that $x \in L_1 \iff f(x) \in L_2$

$$L_1 \leq_p L_2, L_2 \in P \implies L_1 \in P$$

$L \in NPC$

1. $L \in NP$
2. L is NP hard $\forall L' \in NP, L' \leq_p L$ (All NP problems are reducible to L)

   - If we can show that for some $L, L \in P \cup NPC \implies P = NPC$

In [1]:
```
def Quicksort(A, p, r):
    if p < r:
        q = Partition(A, p, r)
        Quicksort(A, p, q-1)
        Quicksort(A,q+1, r)
def Partition(A, p, r):
    # A becomes: [<x..., >x..., undecided..., x]
    # Pivot element is at A[r]

    # To randomize:
    # i = random(p, r)
    # A[i], A[r-1] =

    i=p
    for j in range(p, r):
        if A[j] < A[r]:
            A[i], A[j] = A[j], A[i]
            i+=1
    A[i], A[r] = A[r], A[i]
    return i
```

In [4]:
```
from random import shuffle
x = list(range(100))
shuffle(x)
Quicksort(x, 0, len(x)-1)
sorted(x) == x
```

Out[4]:
```
True
```

In [3]:
```
import numpy as np
def lcs(x,y):
    #make_lcs_len_subseq
    table = np.zeros((len(x)+1,len(y)+1))
    for i in range(len(x)):
        for j in range(len(y)):
            if x[i] == y[j]:
                table[i+1, j+1] = table[i,j] + 1
            else:
                table[i+1, j+1] = max(table[i,j+1], table[i+1, j])
    # trace back lcs subseq
    subseq = ""
    i = len(x)
    j = len(y)
    while table[i,j] != 0:
        if x[i-1] == y[j-1]: #table[i,j] == table[i-1, j-1]
            subseq = x[i-1] + subseq
            i += -1
            j += -1
        elif table[i-1, j] == table[i,j]:
            i += -1
        elif table[i, j-1] == table[i,j]:
            j += -1
        else:
            # shouldn't get here
            print('error')
    return subseq

lcs("spanking", "amputation") == "pain" and lcs('abcuvxy', 'bcwxyz') == '
```

Out[3]:
```
True
```

In [ ]:

In [ ]: