Greedy Algorithms

- A commonly used paradigm for combinatorial algorithms.
- Informally, in “combinatorial” problems, feasible solutions are subsets of discrete input set, so enumerable in exponential time (say, $O(2^n)$). Greedy algorithms find the optimal by searching only a tiny fraction of this space.
- A precise definition is difficult, but informally an algorithm uses “greedy design principle” if it makes a series of choices, and each choice is locally optimal.
- Why should one expect such a myopic strategy to succeed? Indeed, when greedy strategy works, it says something interesting about the structure(nature) of the problem itself!

Making Change

- The coins in US come in four denominations: 25, 10, 5, 1.
- The “change making” problem is to determine how to convert any amount into minimum number of coins.
- Given an integer $X \in \{0, 1, \ldots, 99\}$, find a combination of coins that sum to X using the least number of coins.
- Formally, find integers a, b, c, d with minimum sum (a+b+c+d) so that $X = 25a + 10b + 5c + 1d$

In [25]:

```python
def makeChange(target: int, coins: list) -> list:
    coins.sort(reverse=True)
    numCoins = []

    for coin in coins:
        numCoins.append({"quantity" : target // coin, "coin" : coin})
        target -= target // coin * coin

        if not target:
            break

    if target != 0:
        raise ValueError(
            "Greedy Algorithm cannot make change with target={} and coins={
            .format(target, coins))

    return numCoins

makeChange(73, [25, 10, 5, 1])
```

Out[25]:

```
[{'quantity': 2, 'coin': 25},
 {'quantity': 2, 'coin': 10},
 {'quantity': 0, 'coin': 5},
 {'quantity': 3, 'coin': 1}]
```

Interval Scheduling

- Input: a list of N activities that we want to schedule on a single resource.
- Each activity specified by a start and an end time; only one activity can be scheduled on the resource at a time, and each scheduled activity uses the resource continuously between its start and end time.
- What is the maximum possible number of activities we can schedule?

- Formally, activities is a set $S = \{1, 2, ..., n\}$,where each activity is specified by its start-end time tuple $(s(i), f(i))$, with $s(i) \leq f(i)$.
- This is a combinatorial problem: output is a subset of $\{1, 2, ..., n\}$.
- A feasible schedule is a subset in which no two activities overlap.
- Objective: find a feasible schedule of maximum size (number of activities).

Algorithm

- The correct strategy is to process jobs in the Earliest Finish Time order.
- That is, sort the jobs in the increasing order of their finish time. We assume that jobs are given in this order (by simple relabeling): $f(j_1) \leq f(j_2) \leq f(j_3)... \leq f(j_n)$

Proof of Correctness

- **Lemma:** For any $i \leq k$, we have that $f(a_i) \leq f(b_i)$. (i.e. ith job in greedy finishes no later than the ith job in the optimal.)
- **Proof:**

  1. True for $i = 1$, by the design of greedy.
  2. Inductively assume this is true for all jobs up to $i - 1$, and prove it for i.
  3. The induction hypothesis says that $f(a_{i-1}) \leq f(b_{i-1})$.
  4. Since $f(b_{i-1}) \leq s(b_i)$, we must also have $f(a_{i-1}) \leq s(b_i)$.
  5. So, the ith job selected by optimal is also available to the greedy as its ith job candidate, so whatever job greedy picks it must have $f(a_i) \leq f(b_i)$.
  6. This proves the lemma.
- **Theorem:** The greedy solution is optimal for the activity selection problem.
- **Proof:**

  1. By contradiction. Suppose A is not optimal, and so OPT must have more jobs than A. That is, $m > k$.
  2. Consider what happens when $i = k$ in our lemma. We have that $f(a_k) \leq f(b_k)$. So, the greedy's last job has finished by the time OPT's kth job finishes.
  3. If $m > k$,there is some job that optimal accepts after k,and that job is also available to Greedy; it cannot conflict with anything greedy has scheduled.
  4. Because the greedy does not stop until it no longer has any acceptable jobs left, this is a contradiction.

Runtime

- Sorting the jobs takes $O(nlog(n))$.
- After that, the algorithm makes one scan of the list, spending constant time per job = $O(n)$.
- So total time complexity is $O(nlog(n)) + O(n) = O(nlog(n))$.

In [37]:
```python
def maxActivities(activityList: list) -> dict:
    sortedList = sorted(activityList, key=lambda x: x[1])
    prevEndTime = 0
    activities = list()

    for activity in sortedList:
        if activity[0] >= prevEndTime:
            activities.append(activity)
            prevEndTime = activity[1]

    return {"length" : len(activities), "activities" : activities}

maxActivities([(3,6),(1,4),(4,10),(6,8),(0,2)])
```

Out[37]:
```
{'length': 3, 'activities': [(0, 2), (3, 6), (6, 8)]}
```

Interval Partitioning

- Given a set of activities, schedule them all using a minimum number of machines.

Algorithm

- Sort activities by start time.
- Start Room 1 for activity 1.
- For i = 2 to n, if activity i can fit in any existing room, schedule it in that room.

Proof of Correctness

- Define depth of input set as the maximum number of activities that are concurrent at any time. Let depth be D.
- Optimal must use at least D rooms because a single room can only house 1 activity and there are D concurrent activities that all need different rooms.
- Greedy uses no more than D rooms because a new room is only created when existing rooms are full, meaning the maximum concurrent amount will be the maximum number of rooms created.

Runtime

- Sorting the jobs takes $O(nlog(n))$.
- After that, the algorithm makes one scan of the list, spending a contant operation to check for an open room, and $O(log(n))$ operations to insert the a new room, or replace an existing room = $O(nlog(n))$.
- So total time complexity is $O(nlog(n)) + O(nlog(n)) = O(nlog(n))$.

In [6]:
```python
import heapq

def minPartitions(activityList: list) -> dict:
    if not activityList:
        return 0

    sortedList = sorted(activityList, key=lambda x: x[0])
    endTimes = []
    heapq.heappush(endTimes, sortedList[0][1])

    for i in range(1, len(sortedList)):
        activity = sortedList[i]

        if activity[0] >= endTimes[0]:
            heapq.heappushpop(endTimes, activity[1])
        else:
            heapq.heappush(endTimes, activity[1])

    return {"count": len(endTimes)}

minPartitions([(1,6),(8,13),(15,42),(1,21),(25,31),(35,42)])
```

Out[6]:
```
{'count': 2}
```

Huffman Codes

- Goal: encode characters in as few characters as possible
- With variable encoding length, higher frequency characters can be encoded in shorter bitstrings for higher compression
- Prefix Codes: no codeword can be a prefix of another word
- Encode in a binary tree: characters are leaves and branches are bits (path to leaf is binary encoding)
- Huffman codes are only good at encoding static characters. Dynamic data and words have better encoding methods.

Measuring Optimality

- Let C be the input alphabet (set of distinct characters).
- Let $f(p)$ be the frequency of letter p in C.
- Let T be the tree for a prefix code, and $d_T(p)$ the depth of p in T.
- The number of bits (bit complexity) needed to encode our file using this code is:
$$B(T) = \sum_{p \in C} f(p) d_T(p)$$
- We want a code that achieves the minimum possible value of $B(T)$.

Optimal Tree Property: Tree corresponding to optimal code must be full: that is,each internal node has two children. Otherwise we can improve the code.

Huffman's Algorithm

- The algorithm best understood as building the binary tree T that represents its codes.

- Initially, each letter represented by a single-node tree, whose weight equals the letter's frequency.
- Huffman repeatedly chooses the two smallest trees (by weight), and merges them. The new tree's weight is the sum of the two children's weights.
- If there are n letters in the alphabet, there are $n-1$ merges

Proof of Optimality

- We will use induction on the size of the alphabet $|C|$.
- The base case of $|C| = 2$ is trivial: we have a depth 1 tree, with two leaves, each with code length 1.
- In general, assume induction holds for $|C| = n-1$, and prove for $|C| = n$.
- Take the last two characters x_{n-1} and x_n, combine them into a single new character z with freq. $f(z) = f(x_{n-1}) + f(x_n)$.
- With x_{n-1}, x_n removed and replaced with z, we have a set of size $|C'| = n-1$.
- By induction, we find the optimal code tree of C'. This tree has z at some leaf.
- To obtain tree for C,we attach nodes x_{n-1} and x_n as children of z.
- We will show that given optimal tree for C', this new tree is optimal for C.
- Still one problem: in our construction, the nodes x_{n-1} and x_n will necessarily end upassiblings. (That is, the codes for these two will be identical except in the last bit.)
- How can we choose x_{n-1} and x_n at the onset so that in the optimal tree they are guaranteed to have this property? This is where Huffman's greedy choice enters the proof: we will choose two lowest freq. characters.

Lemma:

- Suppose x and y are two letters of lowest frequency. Then, there exists anoptimal prefix code in which codewords for x and y have the same (and maximum) length and they differ only in the last bit.

Proof:

- Start with an optimal prefix code tree T, and modify it so x and y are sibling leaves of max depth, without increasing total cost.
- In modified tree, x and y have the same code length, different only in the last bit.
- Assume optimal tree does not satisfy the claim, and suppose that a and b are the two characters that are sibling leaves of max depth in T.
- Without loss of generality, assume that $f(a) \leq f(b)$ and $f(x) \leq f(y)$
- We have $f(x) \leq f(a)$ and $f(y) \leq f(b)$. (x, y, a, bneed not all be distinct.)
- First transform T into T' by swapping the positions of x and a
- Since $d_T(a) \geq d_T(x)$ and $f(a) \geq f(x)$, swap does not increase freq * depth cost:
$$
\begin{aligned}
B(T) - B(T') &= \sum_p [f(p)d_T(p)] - \sum_p [f(p)d'_T(p)] \\
&= [f(x)d_T(x) + f(a)d_T(a)] - [f(x)d'_T(x) + f(a)d'_T(a)] \\
&= [f(x)d_T(x) + f(a)d_T(a)] - [f(x)d_T(a) + f(a)d_T(x)] \\
&= [f(a) - f(x)] * [d_T(a) - d_T(x)] \\
&\geq 0
\end{aligned}
$$
- Next, transform T' into T'' by exchanging y and b, which also does not increase cost.
- So, we get that $B(T'') \leq B(T') \leq B(T)$. If T was optimal, so is T'', but in $T''x$ and y are sibling leaves at the max depth.

Proof of optimality:

- Let T_1 be the optimal tree (induction) for $C + \{z\} - \{x, y\}$.
- We obtain our final tree T by attaching leaves x, y as children of z.
- What is the connection between costs of $B(T)$ and $B(T_1)$?
- For all $p \neq x, y$ depth is the same in both trees, so no difference. For x, y, we have $d_T(x) = d_T(y) = d_{T_1}(z) + 1$. So, the cost increase from modifying T_1 to T is: $B(T) - B(T_1) = f(x) + f(y)$ because
$$f(x)d_T(x) + f(y)d_T(y) = [f(x) + f(y)] * [d_{T_1}(z) + 1] = f(z)d_{T_1}(z) + [f(x) + f(y)]$$
- The rest of the argument is via contradiction.
- Suppose T is not an optimal prefix code, and another tree T_0 is claimed to be optimal, meaning $B(T_0) < B(T)$.
- By previous lemma, T_0 has x and y as siblings. Imagine replacing parent of x, y with a new leaf z,with freq. $f(z) = f(x) + f(y)$, and call this new tree T_1'.
- Then, $B(T_1') = B(T') - f(x) - f(y) < B(T) - f(x) - f(y) < B(T_1)$ which contradicts the claim that T_1 is an optimal prefix code for $C' = C + \{z\} - \{x, y\}$.

Time Complexity

- Time complexity is $O(nlogn)$. Initial sorting plus n heap operations.

In [ ]:
```
# Insert Code
```

Horn Formulas

- Form of boolean logic, and often used in AI systems for logical reasoning.
- Each boolean variable represents an event (or possibility), such as
- x = the murder took place in the kitchen
- y = the butler is innocent
- z = the colonel was asleep at 8pm.
- Recall that Boolean variable can only take one of two values $\{true, false\}$, and a literal is either a variable x or its negation $\overline{x}$

Constraints among variables represented by two kinds of clauses:

1. Implication: Left-hand-side is an AND of any number of positive literals, and right-hand-side is a single positive literal. $(z \cap u) \rightarrow x$ It asserts that “if the colonel was asleep at 8 pm, and the murder took place at 8pm, then the murder took places in the kitchen.” A degenerate statement of the type $\rightarrow x$ means that x is unconditionally true. For instance, “the murder definitely occurred in the kitchen.”
2. Negative: Consists of an OR of any number of negative literals, as in $(\overline{u} \cup \overline{t} \cup \overline{y})$, where $u, t, y, resp.$, means that constable, colonel, and butler is innocent. This clause asserts that “they can’t all be innocent.”

- A Horn formula is a set of implications and negative clauses.
- Problem: Given a Horn formula, decide if it is satisfiable, namely, is there an as-signment of variables so that all clauses are satisfied. Such an assignment is called asatisfying assignment.

Examples:

- The Horn formula $\rightarrow x, \rightarrow y, x \cap u \rightarrow z, \overline{x} \cup \overline{y} \cup \overline{z}$ has a satisfying assignment $u = 0, x = 1, y = 1, z = 0$.
- But the formula $\rightarrow x, \rightarrow y, x \cap y \rightarrow z, \overline{x} \cup \overline{y} \cup \overline{z}$ is not satisfiable.

Algorithm

- Brute force approach would take 2^n to account for powerset of inputs.
- The nature of Horn clauses suggests a natural greedy algorithm:
- Initially set all variables to false.
- While there is an unsatisfied Implication clause, set its RHS to true.
- If all pure negative clauses are satisfied, return the assignment; otherwise, formula is not satisfiable.

Correctness Proof

- Clearly, if the algorithm returns a satisfying assignment, then it is a valid assignment because it satisfies all negative and implication clauses.
- To show that if the algorithm does not find a satisfying assignment, there is none, we observe that the algorithm maintains the following invariant. If a certain set of variables is set to true, then they must be true in any satisfying assignment. Namely, we only set a variable true when it is forced upon us.

Time Complexity

- With some care the greedy algorithm can be implemented in linear time (in the length of the formula).

In [ ]:
```
# Insert Code
```

Set Cover

- Input is a (ground) set of n elements $B = \{1, 2, \ldots, n\}$ and a collection of m subsets $S = \{S_1, S_2, \ldots, S_m\}$, with each $S_i \subseteq B$.
- The problem is to choose the smallest number of subsets whose union is B.
- Example: $B = \{1, 2, 3, 4, 5\}$, and $\{\{1, 2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$. One can cover all items by choosing all four sets, but sets $\{1, 2, 3\}, \{4, 5\}$ suffice.

Algorithm

- Repeat until all elements of B are covered: pick the set S_t containing the largest number of still-uncovered elements.

Runtime

- If the optimal solution uses k sets, the greedy uses $O(kln(n))$ sets.

In [ ]:
```
# Insert Code
```

Dijkstra’s Algorithm

1. Let S be the set of explored nodes.
2. Let $d(u)$E be the shortest path distance from s to u, for each $u \in S$.
3. Initially $S = \{s\}, d(s) = 0$, and $d(u) = 1$, for all $u \neq s$.
4. While $S \neq V$ do
5. Select $v \notin S$ with the minimum value of $d'(v) = \min_{(u,v),u \in S} d(u) + cost(u, v)$
6. Add v to S,set $d(v) = d'(v)$.

Correctness Proof

1. Argue that at any time $d(v)$ is the shortest path distance to v, for all $v \in S$.
2. Consider the instant when node v is chosen by the algorithm. Let (u, v) be the edge, with $u \in S$, that is incident to v.
3. Suppose, for the sake of contradiction, that $d(u) + cost(u, v)$ is not the shortest path distance to v. Instead a shorter path P exists to v.
4. Since that path starts at s,it has to leave S at some node. Let x be that node, and let $y \notin S$ be the edge that goes from S to $\overline{S}$.
5. So our claim is that $length(P) = d(x) + cost(x, y) + length(y, v)$ is shorter than $d(u) + cost(u, v)$. But note that the algorithm chose v over y, so it must be that $d(u) + cost(u, v) \leq d(x) + cost(x, y)$.
6. In addition, since $length(y, v) > 0$, this contradicts our hypothesis that P is shorter than $d(u) + cost(u, v)$.
7. Thus, the $d(v) = d(u) + cost(u, v)$ is correct shortest path distance.

In [ ]:
```
# Insert Code
```

Kruskal's Algorithm

1. If the shortest edge connects two previously unconnected vertices, add that edge to the spanning tree.
2. Continue repeating step 1 until all the vertices are connected.

Correctness Proof

1. For simplicity, assume that all edge costs are distinct so that the MST is unique.Otherwise, add a tie-breaking rule to consistency order the edges.
2. Proof by contradiction: let (v, w) be the first edge chosen by Kruskal that is not in the optimal MST.
3. Consider the state of the Kruskal just before (v, w) is considered.
4. Let S be the set of nodes connected to v by a path in this graph. Clearly, $w \notin S$.

5. The optimal MST does not contain (v, w) but must contain a path connecting v to w,by virtue of being spanning.
6. Since $v \in S$ and $w \notin S$, this path must contain at least one edge (x, y) with $x \in S$ and $y \notin S$.
7. Note that (x, y) cannot be in Kruskal's graph at the time (v, w) was considered because otherwise y will have been in S.
8. Thus, (x, y) is more expensive than (v, w) because it came after (v, w) in Kruskal's scan order.
9. If we replace (x, y) with (v, w) in the optimal MST, it remains spanning and has lower cost, which contradicts its optimality.
10. So, the hypothesis that (v, w) is not in optimal must be false.

In [ ]:
```
# Insert Code
```

Divide and Conquer Algorithms

- A general paradigm for algorithm design; inspired by emperors and colonizers.

1. Divide the problem into smaller problems.
2. Conquer by solving these problems.
3. Combine these results together.

Binary Search

- Search for x in sorted array A.
- If x is equal to the middle element of A, search is complete
- If x is less than the middle element of A, search on the left half of A
- Else, search on the right half of A

Time Complexity

- Let $T(n)$ denote the worst-case time to binary search in an array of length n.
- Recurrence is $T(n) = T(n/2) + O(1)$.
- $T(n) = O(logn)$

In [2]:
```
def binarySearch(target: int, arr: list, left: int, right: int) -> int:
    if left > right:
        return -1

    middle = (left + right) // 2
    if target == arr[middle]:
        return middle
    elif target < arr[middle]:
        return binarySearch(target, arr, left, middle - 1)
    else: #target > arr[middle]
        return binarySearch(target, arr, middle + 1, right)

print(binarySearch(-1, list(range(10)), 0, 9))
print(binarySearch(10, list(range(10)), 0, 9))
print(binarySearch(5, list(range(10)), 0, 9))
```

Merge Sort

- Sort an unsorted array of numbers A
- If array is one element, return A
- Otherwise, recursively call mergesort on the left and right halves of A
- Then, merge the sorted result of the left and right haves of A

Time Complexity

- Let $T(n)$ denote the worst-case time to merge sortan array of length n.

- Recurrence is $T(n) = 2T(n/2) + O(n)$.
- $T(n) = O(nlogn)$

```
In [ ]: # CODE
```

Multiplying Numbers

- We want to multiply two n-bit numbers. Cost is number of elementary bit steps.
- Grade school method has $O(n^2)$ cost: n^2 multiplies, $n^2/2$ additions, plus some carries.

Karatsuba's Algorithm

- Let X and Y be two n-bit numbers. Write $X = ab, Y = cd$ where ab and cd are concatenated to form an n-bit number.
- a, b, c, d are $n/2$ bit numbers. (Assume $n = 2^k$.)
$$XY = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd$$
- Note that $(a - b)(c - d) = (ac + bd) - (ad + bc)$.
- Solve 3 subproblems: $ac, bd, (a - b)(c - d)$.
- We can get all the terms needed for XY by addition and subtraction!

Time Complexity

- The recurrence for this algorithm is $T(n) = 3T(n/2) + O(n) = O(n^{log_2(3)})$.
- The complexity is $O(n^{log_2(3)}) = O(n^{1.59})$.

```
In [ ]: # CODE
```

Recurence Solving

- Expand terms until a general formula is reached.
- Substitute for base case and solve.
- Can also use tree view with number of levels and work per level.
- Can solve by induction.

Master Method

- Recurrence in the form
$$T(n) = O(n^{log_b(a)}) + \sum_{i=0}^{log_b(n-1)} a^i f(\frac{n}{b^i})$$
- Let $f(n) = O(n^p log^k(n))$ where $p, k \geq 0$
- Condition: $a \geq 1, b > 1$ must be constant
- Case 1: $p < log_b a$ => $n^{log_b(a)}$ grows faster than $f(n)$. Thus, $T(n) = O(n^{log_b(a)})$.
- Case 2: $p = log_b a$ => both terms have same growth rates, thus $O(n^{log_b(a)} log^{k+1}(n))$
- Case 3: $p > log_b a$ => $n^{log_b(a)}$ grows slower than $f(n)$. Thus, $T(n) = O(f(n))$

Matrix Multiplication

- Multiply two $n \times n$ matrices: $C = A \times B$.

Traditional Algorithm

- Standard Method: $C[i][j] = \sum_{k=1}^{n} A[i][k] \times B[k][j]$
- For every element in C, it takes $O(n)$ computations.
- There are n^2 elements in C so it takes $O(n^3)$.

Strassen's Algorithm

- Let A, B be two $n \times n$ matrices.
- Divide matrices A, B, C into four $n/2 \times n/2$ submatrices.
$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}; B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}; C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$
- We can rewrite the product matrices as the following:
$$\begin{aligned} c_{11} &= a_{11} * b_{11} + a_{12} * b_{21} \\ c_{12} &= a_{11} * b_{12} + a_{12} * b_{22} \\ c_{21} &= a_{21} * b_{11} + a_{22} * b_{21} \\ c_{22} &= a_{21} * b_{12} + a_{22} * b_{22} \end{aligned}$$
- However, the recurrence for this relation listed below solves to $O(n^3)$:
$$T(n) = 8T(n/2) + O(n^2)$$
- Can reduce to seven multiplications using the following matrices:
$$\begin{aligned} P_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ P_2 &= (a_{21} + a_{22})(b_{11}) \\ P_3 &= (a_{11})(b_{12} - b_{22}) \\ P_4 &= (a_{22})(b_{21} - b_{11}) \\ P_5 &= (a_{11} + a_{12})(b_{22}) \\ P_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ P_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \end{aligned}$$
- We can rewrite the product matrices as the following:
$$\begin{aligned} c_{11} &= P_1 + P_4 - P_5 + P_7 \\ c_{12} &= P_3 + P_5 \\ c_{21} &= P_2 + P_4 \\ c_{22} &= P_1 + P_3 - P_2 + P_6 \end{aligned}$$
- The recurrence for this relation listed below solves to $O(n^{\log_2(7)}) = O(n^{2.81})$:
$$T(n) = 7T(n/2) + O(n^2)$$

Quicksort

- Simple, fast, and does not require extra space

Algorithm

- Partition among a pivot, splitting into elements smaller than the pivot, denoted L, and elements greater than the pivot, denoted R
- Sort L and R recursively
- Combine by appending R to L

Time Complexity

- $T(n)$ denotes the randomized runtime of Quicksort
- Each element randomly likely to be chosen as a pivot so there is $1/n$ probability that i is the pivot.
- Recurrence denoted by the following relation:

$$T(n) = 1/n * \sum_{i=1}^{n}(T(i-1) + T(n-1)) + n + 1$$

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

$$T(n) = 2/n * \sum_{i=0}^{n-1} T(i) + n + 1$$

$$(1) : n * T(n) = 2 * \sum_{i=0}^{n-1} T(i) + n^2 + n$$

$$(2) : (n-1) * T(n-1) = 2 * \sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1)$$

- Subtract (2) from (1) to arrive at the following:

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

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

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

$$\frac{T(n)}{n+1} = \frac{T(2)}{3} + \sum_{i=3}^{n} \frac{2}{i}$$

$$\frac{T(n)}{n+1} = O(1) + 2\ln(n)$$

- Thus, $T(n) \leq 2(n+1)\ln(n)$, which is linearithmic.

In [ ]:
```
# CODE
```

Extrema Finding

- We can find the maximum and minimum in linear time with n comparisons.
- We can divide and conquer to find both the min and max in 3n/2 comparisons.

Min Algorithm

- Initialize current minimum to be the first element.
- Iterate through the rest of the elements; if any element is less than the current minimum, set it as the new current minimum.

Min Max Algorithm

- If the list A contains a single element, $min = max = A[0]$.
- Divide into two equal sublists A_1, A_2 and recursively find both the min and the max of both sublists. Then, return the more extreme of the two results for each min and max.

Time Complexity

- 2 calls on half the list + 2 comparisons has a recurrence of the following:
$$T(n) = 2T(n/2) + 2$$

Using the recurrence expansion method, we get...

$$T(n) = 2 * (2 * T(n/2^2) + 2) + 2 = 2^2 * T(n/2^2) + 2^2 + 2$$
$$T(n) = 2^2 * (2 * T(n/2^3) + 2) + 2^2 + 2 = 2^3 * T(n/2^3) + 2^3 + 2^2 + 2$$

...

$$T(n) = 2^i * T(n/2^i) + 2^i + ... + 2 = 2^i * T(n/2^i) + 2(2^{i-1} + ... + 2 + 1)$$
$$T(n) = 2^i * T(n/2^i) + 2(2^i - 1) = 2^i * T(n/2^i) + 2 * 2^i - 2$$

Use $T(2) = 1$. Then $n/2^i = 2$ when $i = \log_2 n/2$

Substitute i to get the recursion $T(n) = n/2 + 2 * n/2 - 2 = 3n/2 - 2$

In [3]:
```python
def findMin(l: list) -> float:
    minimum = l[0]
    for element in l[1:]:
        minimum = element if element < minimum else minimum
    return minimum
print(findMin(list(range(10, 0, -1))))
```

```
1
```

In [6]:
```
def minMax(l: list) -> tuple:
    if len(l) == 1:
        return (l[0], l[0])
    elif len(l) == 2:
        return (l[0], l[1]) if l[0] < l[1] else (l[1], l[0])
    else:
        half = len(l) // 2
        min1, max1 = minMax(l[:half])
        min2, max2 = minMax(l[half:])
        minimum = min1 if min1 < min2 else min2
        maximum = max1 if max1 > max2 else max2
        return (minimum, maximum)
print(minMax(list(range(20))))
```

```
(0, 19)
```

Linear Time Selection

- Find the item of rank k in the list (indexed 1 as smallest and n as largest).

Algorithm

- Divide items into $n/5$ groups of 5 each.
- Find the median of each group using sorting.
- Recursively find median of $n/5$ group medians.
- Partition using median-of-median, x, as a pivot.
- Let low side have s items and high side have $n - s$ items. If $k \leq s$, call this algorithm on the low side. Else, call this algorithm on the high side for rank $k - s$.

Correctness Proof

- The base case is trivial.
- If we call the low side, when $k \leq x$, we consider all items not in the quadrant greater than x. We use the inductive hypothesis to assume this recursion returns the correct result.
- Without loss of generality, we can apply this to the high side as well.

Time Complexity

- Recursively finding the group median is a recursive call of $T(n/5)$.
- Recrusively calling the low or high side is a recursive call of $T(7n/10)$ as there are $1/2 * n/5$ groups contributing at least 3 items to the opposite side.
- All other work can be done in linear time.
- The recurrence relation is the following:
$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$
- We can inductively verify $T(n) \leq cn$ for some constant c:
$$T(n) \leq c(n/5) + c(7n/10) + O(n)$$
$$T(n) \leq (9/10)cn + O(n) \leq cn$$
$$T(n) \leq O(n) \leq cn/10$$
- Choose c so that $cn/10$ beats $O(n)$ for all n. Thus, $T(n) \leq cn$, meaning it runs in linear time.

In [ ]:
```
# CODE
```

Convex Hulls

- Smallest convex shape that contains a set of points

Algorithm

- Sort points by x-coordinates.
- Partition points into equal halves A (left) and B (right).
- Recursively compute the convex hull of A and B.
- Merge the convex hulls of A and B to arrive at the overall convex hull: start at the rightmost point a of A and leftmost point b of B; while a, b is not the lower tangent of the convex hulls of A and B: move A clockwise around points of A until it is a tangent of A, move b counter clockwise until it is a tangent of B. Then, repeat the process for the upper tangent in the reverse direction. Remove edges that were travelled in the rotation.

Correctness Proof

- Tangent of both objects does not cutoff any point
- Tangent of both objects also does not add any additional unnecessary space
- We explicitly check for tangent of both sides and remove unnecessary edges

Time Complexity

- Initial sorting takes $O(n \log(n))$.
- Recurrence = $T(n) = 2T(n/2) + O(n)$ with $O(n)$ for tangent merging.
- Recurrence solves to $O(n \log(n))$.

In [ ]:
```
# CODE
```