ELEC50006: Discrete Mathematics (Goodman)

Clemen Kok — Imperial College London

April 3, 2023

Contents

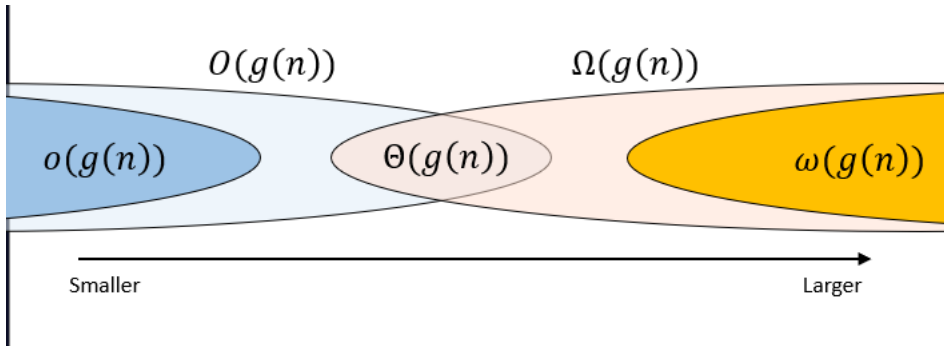


Figure 1: Scale of Complexities

1 Complexity Analysis

1.1 Asymptotic Notation

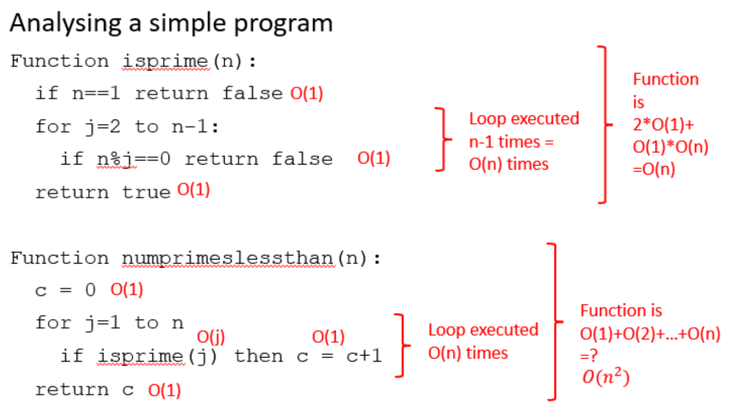
Introduction. Complexity Notation: f(n) is O(g(n)) - f(n) could be the number of steps a program takes to run, the time a program takes to run or the amount of memory a program uses.

Theta. f(n) is θ(g(n)) means that f(n) is *approximately equal* to g(n), ignoring constant factors. O means less than θ (program takes at most this long to run) and Ω means greater than θ (program takes at least this long to run).

Rigorous Definition. A function f(n) is θ(g(n)) if there are constants $c_1 > 0$, $c_2 > 0$ and n_0 such that for all n $\geq n_0$, $0 < c_1 \leq \frac{f(n)}{g(n)} \geq c_2$. Technically correct but rare: f $\in \theta$(g(n)); Incorrect but often used: f(n) = θ(g(n)).

O and Omega. A function f(n) is O(g(n)) if there are constants $c > 0$ and n_0 such that for all n $\geq n_0$, $0 \leq f(n)/g(n) \leq c$. A function f(n) is Ω(g(n)) if there are constants c ¿ 0 and n_0 such that for all n $\geq n_0$, $0 < c \leq f(n)/g(n)$. f(n) is only θ(g(n)) iff it is O(g(n)) and Ω(g(n)).

Little O (o) and Little Omega (ω). Used less often, and stronger than O and Ω. A function is f(n) is o(g(n)) if for any constant $c > 0$ and there is an n_0 such that for n $\geq n_0$, $0 \leq f(n)/g(n) < c$. A function is ω(g(n)) if for any constant $c > 0$ there is an n_0 such that for all $n \geq n_0, 0 < c < f(n)/g(n)$.



Analysing a simple program

```
Function isprime(n):
  if n==1 return false
  for j=2 to n-1:
    if n%j==0 return false
  return true

Function numprimeslessthan(n):
  c = 0
  for j=1 to n
    if isprime(j) then c = c+1
  return c
```

O(1)
O(1)
O(1)
Loop executed n-1 times = O(n) times
Function is 2*O(1)+ O(1)*O(n) =O(n)
O(1)
O(j)
O(1)
O(1)
Loop executed O(n) times
Function is O(1)+O(2)+...+O(n) =? $O(n^2)$

Figure 2: Simple Program Analysis

1.2 Calculating with Complexities

Polynomials. We only need the highest power. Example: : $2n^3 + 5n^2 + 11n + 2$ is $O(n^3)$. Constants < Logarithms < Polynomials < Exponentials.

1. O(1) - fixed amount of time regardless of n. Assume arithmetic is O(1). Example algorithm: Lookup.
2. O(log(n)) - not much bigger than 1. Doubling n increases log(n) by a small amount. log(n) is smaller than $n^{anything}$. Example algorithm: Binary Search in Sorted Array.
3. O(n) and O(nlogn) - very common. Example algorithms: O(n) - unsorted search, sum of array; O(nlogn) - sorting, FFT. Soft-O Notation: nlogn = $\tilde{O}$(n).
4. $O(n^2), O(n^3)$ - bad but sometimes unavoidable. Example - convolution, matrix multiplication.
5. $O(2^n), O(n!), O(n^n)$ - sometimes we have to use these even for small n.

Summations. If f(n) = θ(n) then

$$\sum_{j=1}^{n} f(j) = \theta(n^2)$$

2 Divide & Conquer

2.1 Algorithm

Introduction. Divide problem into subproblems. Conquer subproblems independently. Combine Solutions. Worst Case Complexity - O(nlogn).

Merge Sort. Naive sort is $\theta(n^2)$. In merge sort, we (1) divide two equal subarrays, (2) conquer (sort) these subarrays, and (3) combine (merge) sorted subarrays into sorted array. Combining is $\theta(n)$ with joint iteration.

Code:

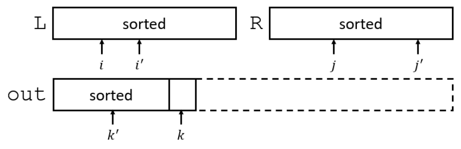
```python
def mergesort(X):
    if len(X) <= 1:
        return X
    mid = len(X) // 2
    L = mergesort(X[:mid])
    R = mergesort(X[mid:])
    out = []
    i = j = 0
    while i<len(L) or j<len(R):
        if i==len(L):
            out.append(R[j])
            j += 1
        elif j==len(R):
            out.append(L[i])
            i += 1
        else:
            l = L[i]
            r = R[i]
            if l < r:
                out.append(L[i])
                i += 1
            else:
                out.append(R[j])
                j += 1
    return out

mergesort([6, 5, 3, 1, 8, 7, 2, 4])
```

Proving merge sort is correct with loop invariants. Proof by induction on iteration index of algorithm. *Loop Invariance* is a key property of the state of a program that has three qualities: (1) Initialisation: True at beginning of first iteration, (2) Maintenance: True at iteration k implies true at $k+1$, (3) Termination: True at end of loop proves what we want. Merge sort is loop invariant if (1) `out` is sorted, and (2) All items in `out` are smaller than all remaining in L and R. We want property (1) to be true at the end of the loop.

The simple explanation. At the start of each iteration of the merge loop, the sub-lists being merged are already sorted. This loop invariance is maintained by the merging algorithm, which compares the first element of each sub-list and selects the smaller one to add to the merged list. Since both sub-lists are already sorted, the smallest element is guaranteed to be the first element of one of the sub-lists. As each element is added to the merged list, the algorithm advances the corresponding sub-list by one position. Because the sub-lists were sorted at the beginning of the iteration, and only the smallest element was removed and added to the merged list, the loop invariant is maintained. Once the merge loop has completed, the resulting merged list is also guaranteed to be sorted, since it was built by repeatedly selecting the smallest element from two already-sorted sub-lists. Therefore, we can conclude that merge sort maintains the loop invariant that at the start of each iteration of the merge loop, the sub-lists being merged are already sorted.

Figure 3: Merge Sort Loop Invariance



To show that `out` remains sorted. We need to show that the new element of `out` is larger than all previous ones. We prove by contradiction. If (1) is not true, then: $\exists k'$ with $k' < k : out[k] < out[k']$. But, $out[k] = L[i](or, R[j])$. So, $L[i] < out[k']$. This contradicts the assumption that all items in out are smaller than all remaining items in L and R with i' = i. So the assumption that `out` is sorted $\forall k_1, k_2$ with $0 \leq k_1 < k_2 < |out| : out[k_1] \leq out[k_2]$.

To show that all items of `out` is smaller than all remaining in L, R. We need to show this only for the last element, out[k]. We know that out[k] = min { L[i], R[j] }. This means that out[k] ≤ L[i] and out[k] ≤ R[j]. L is sorted, so: $\forall i'$ with $i' \geq i : L[i] \leq L[i']$ and therefore out[k] ≤ L[i] ≤ L[i'] so $\forall i'$ with $i' \geq i : out[k] \leq L[i']$ so (2L) is true, vv. for (2R). (2L) = $\forall k', i'$ with $0 \leq k' < |out|, i \leq i' < |L| : out[k'] \leq L[i']$.

Merge Sort is Theoretically Optimal. n! possible orderings of n items. The algorithm needs to determine which order the array is in. Each comparison at best halves the number of remaining possible orderings. After k comparisons, $n!/2^k$ remaining possible comparisons. So, we need $log_2 n!$ comaprisons to get to 1 possible. $logn! = \Omega(nlogn)$. So, sorting with comparisons is $\Omega(nlogn)$.

2.2 Master Method

Introduction. If, for some $a > 0, b > 0$ and $d \geq 0$, $T(n) = aT(n/b) + O(n^d)$, then

$$T(n) = \begin{cases} O(n^d), & \text{if } d > log_b a. \\ O(n^d logn), & \text{if } d = log_b a. \\ O(n^{log_b a}), & \text{if } d < log_b a \end{cases} \tag{1}$$

Origin of Master Method. Total Work = Work done at last step + Work done combining. Work done at last step = O(Work done combining). Total Work = O(Work done combining). Work done combining = $n^d \frac{a^k - 1}{a-1}$, giving three cases: a < 1, a > 1 or a = 1.

2.3 Special Problems

Maximum Subarray. When do we buy and sell in a stock market? We are only given price changes, so we transform the problem to find the maximum sum of differences. This gives us a brute force solution of $\theta(n^3)$ or $\theta(n^2)$. According to the maximum subarray problem, we need to find i and j so that X[i] + X[i+1] + ... X[j] is maximum. We can do this via Divide and Conquer. The number of steps is T(n) = 2T(n/2) + O(n), so its the same as merge sort; T(n) = O(nlogn).

Code:

```python
def dac_maxsubarray(X):
    # Base Case
    if len(X) <=1:
        return sum(X), 0, len(X)
    # Divide
    mid = len(X) / 2
    # Conquer Left
    left_sum, left_i, left_j = dac_maxsubarray(X[:mid])
    # Conquer Right
    right_sum, right_i, right_j = dac_maxsubarray(X[mid:])

    # Correct for relative indices
    right_i += mid
    right_j += mid
    cursum = mid_sum = 0
    mid_i = mid_j = mid
    # Start from middle and move left
    for i in range (mid-1, -1, -1):
        cursum += X[i]
        if cursum > mid_sum:
            mid_sum, mid_i = cursum, i
    cursum = mid_sum
    # Start from middle and move right
    for j in range(mid+1, len(X)+1):
        cursum += X[j-1]
        if cursum > mid_sum:
            mid_sum, mid_j = cursum, j
    # Combine
    if left_sum >= right_sum and left_sum >= mid_sum:
        return left_sum, left_i, left_j
    elif right_sum >= mid_sum:
        return right_sum, right_i, right_j
    else:
        return mid_sum, mid_i, mid_j
```

Large Number Multiplication. Naive Algorithm: Long Multiplication - cost is $O(n^2)$. DAC nets complexity equation of $T(n) = 4T(n/2) + O(n)$ which gives $O(n^2)$ according to Master Method. *Karatsuba's Trick* nets $O(n^1.58)$ according to Master Method.

3 Dynamic Programming

3.1 Overview

Introduction to Dynamic Programming. In Divide and Conquer, we divide the problem into independent subproblems, conquer those and combine. In Dynamic Programming, we treat it as an optimisation problem where we break the problem into a series of choices. Each choice leads to an overlapping subproblem, and we typically solve these recursively and using memoisation.

Memoisation. In recursive fibonnaci, we have to recompute the same value over and over again. The solution is to store previously computed values - this is a technique called *memoisation*.

Optimal Substructure. The optimal solution to the whole problem is a combination of optimal solutions to subproblems. The strategy is to iterate through all ways of breaking into subproblems, recurisvely solve these using memoisation and return the best solution from these. We can use this to reduce exponential problems into polynomial time.

3.2 Problems

Rod Cutting. A factory produces 8m long rods and sells it for £20. Its competitor is selling 4m long rods for £15. By cutting the rods in half, our price goes up from £20 to £30. Now, given a pricing table of:

Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

We can cut into 2m and 6m long rods, making £22. But is this the most optimised cut?

Dynamic Programming Solution. What's the best price V_n? We start by making a cut of length i. This gives price $p_i + V_{n-i}$, so the best price we can get is $V_n = \max p_i + V_{n-i}$. The algorithm to compute V_n is to iterate over all i=1 to n-1, recursively compute the price V_{n-i}, memoising the results, then choosing the i that gives the largest value of $p_i + V_{n-i}$. To commpute V_n, we have to iterate from 1 to n. This takes $f(n) = \theta(n)$ steps. Eventually, we will compute all $V_1, V_2 ... V_n$ so $\sum_{n}^{j=1} f(j) = \theta(n^2)$ steps. This reduces the time complexity from $\Omega(2^n)$ to $\theta(n^2)$.

Reconstruction. What cuts should we make? Easy to change: just memorise the list of cuts as well as best price. Memory usage (worst case): for length n, the lsit of cuts [1,1 ... 1] uses f(n) = O(n) memory. We have to store the solution for all rod lengths - so f(1) + f(2) + ... f(n) = $O(n^2)$. We can do better - we currently store the same information repeatedly. Thus, we can only store one of the cuts (doesn't matter which one) and reconstruct the solution at the end. The memory cost is O(n) and the reconstruction cost is O(n), which doesn't change the total.

Code:

```
""" This code implements a dynamic programming solution for the rod-cutting problem, which aims to find the optimal way to cut a rod of length n into smaller pieces that can be sold for different prices.

The rod_cut_table function is a recursive function that uses memoization to avoid recomputing the same subproblems. It takes three arguments: n, the length of the remaining rod; p, a list of prices for each possible cut; and memo, a dictionary that stores the best price and best cut for each length of rod that has already been solved. The function first checks if the solution for rod of length n is already in the memo, if not, it computes the best way to cut the rod using a loop over all possible cuts, recursively calling rod_cut_table on the remaining part of the rod and adding the price of the current cut. The function then updates the memo with the best price and cut for the current length of rod and returns them.

The rod_cut function is the main function that takes two arguments: n, the length of the rod to be cut; and p, a list of prices for each possible cut. The function initializes the memo with solutions for rods of length 0 and 1, and then calls rod_cut_table to solve the problem for rod of length n. It then constructs a list of the best cuts by repeatedly retrieving the next best cut from the memo until the length of the rod becomes 0. Finally, it returns the best price and the list of cuts.

The code defines a list p of prices for cuts of different lengths,
```

```python
    and then calls rod_cut with an argument of 8 to find the best
    way to cut a rod of length 8. The output should be a tuple of
    the best price and a list of cuts, which represents the optimal
     way to cut the rod to maximize the profit.
 """

def rod_cut_table (n, p, memo):
    if n not in memo:
        bestprice = 0
        for i in range(1, n+1):
            price, cut = rod_cut_table(n-i, p, memo)
            price += p[i]
            if price > bestprice:
                bestprice = price
                bestcut = i
        memo[n] = (bestprice, bestcut)
    return memo[n]

def rod_cut(n, p):
    memo = {0: (0, 0),
            1: (p[1], 1)}
    bestprice, bestcut = rod_cut_table(n, p, memo)
    best_cuts = []
    while n:
        price, nextcut = memo[n]
        best_cuts.append(nextcut)
        n -= nextcut
    return bestprice, best_cuts

p = [0, 1, 5, 8, 9, 10, 17, 17, 20]
print(rod_cut(8,p))
```

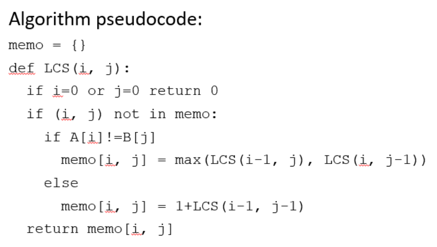
Longest Common Subword. Find the longest common subword from two strings (e.g. aabaababaa and abababaaabb gives abababaa). The general problem is A = $a_1a_2...a_n$ and B = $b_1b_2...b_m$. The brute force solution is that the number of subwords of A is 2^n. Checking if a word is a subword of B costs O(m), so the overall time complexity is $O(2^nm)$.

Dynamic Programming Solution. The length of the Longest Common Subword between A_i and B_j is p(i,j). To calculate p: if $a_i \neq b_j$ then the LCS of A_i and B_j must be the LCS of A_{i-1} and B_j or the LCS of A_i or B_{j-1}. So p = max{p(i-1,j),p(i,j-1)}. If $a_i = b_j$ then the LCS of A_i and B_j must be the LCS of A_{i-1} and B_{j-1} with $a_i = b_j$ appended. So p(i,j) = p(i-1,j-1)+1. Each function call is O(1) and we compute each p(i,j) at most once. So the overall time complexity is O(mn).

Code (2D Dynamic Programming):

```python
""" This code is implementing the dynamic programming approach to
    find the length of the longest common subsequence between two
    given strings, text1 and text2.
```

Figure 4: Psuedocode for Longest Common Subword



Algorithm pseudocode:

```
memo = {}
def LCS(i, j):
  if i=0 or j=0 return 0
  if (i, j) not in memo:
    if A[i]!=B[j]
      memo[i, j] = max(LCS(i-1, j), LCS(i, j-1))
    else
      memo[i, j] = 1+LCS(i-1, j-1)
  return memo[i, j]
```

```python

The function takes two string inputs and returns an integer which represents the length of the longest common subsequence.

The code uses two nested loops to iterate through each character of the two strings from the end to the beginning. The 'dp' variable in the code represents a two-dimensional list which is used to store the length of the longest common subsequence at each position.

If the characters in both strings match, the value of dp[i][j] is set to 1 plus the value of dp[i+1][j+1]. If the characters do not match, the value of dp[i][j] is set to the maximum value between dp[i][j+1] and dp[i+1][j].

Finally, the function returns the value of dp[0][0], which represents the length of the longest common subsequence between the two input strings.
 """

def longestCommonSubsequence(self, text1: str, text2: str) -> int:
    for i in range(len(text1)-1,-1,-1):
        for j in range(len(text2)-1,-1,-1):
            if text1[i] == text2[j]:
                dp[i][j] = 1 + dp[i+1][j+1]
            else:
                dp[i][j] = max(dp[i][j+1], dp[i+1][j])
    return dp[0][0]
```

4 Greedy Algorithms

4.1 Overview.

Greedy Algorithms. Greedy Algorithms represent a set of optimisation problems where we have a series of choices; we need to make the best choice without looking at alternatives. This requires the *greedy choice property.* Comparatively, in Dynamic Programming, we consider all choices to find which is best.

Hungry Student Problem. Each student who attends a whole lecture is given cake. Students know the timetable of all lectures. What is the best strategy to maximise the amount of cake obtained? To solve: in Dynamic Programming, we have optimal substructure - the choice of the first lecture to go to creates a subproblem. In the Greedy Algorithm, we identify the greedy choice property. There are no trade-offs to consider in this choice, so we choose the option that maximises the remaining options and choose the lecture that finishes earliest.

Greedy Solution to Hungry Student Problem. We (1) sort lectures by finishing time, (2) select the lecture that finishes first, (3) remove lectures that start before this time, then (4) go back to 2 and repeat until none remain. Complexity: Sorting - O(nlogn), other steps - O(n), so overall complexity is O(nlogn).

Code:

```python
def greedy_student(events):
    # events is a list of triples (name,start,end)
    # this code sorts triples by their third item
    events.sort(key=lambda(name,start,end: end))
    last_end_time = 0
    schedule = []
    for name, start, end in events:
        if start >= last_end_time:
            schedule.append(name)
            last_end_time = end
    return schedule

print greedy_student([("Spanish 101",0,3), ("Introduction to Film"
    ,1,3), ("Social Psychology",2,3), ("Advanced Criminal Law",1,2)
    ])
```

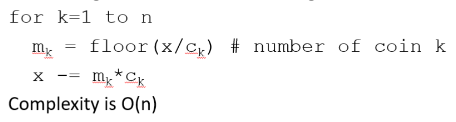
Proving Greedy Choice. We want to prove that the lecture that finishes earliest is in an optimal choice of lectures. We assume that the set C is an optimal set of choices, and that the earliest-to-finish lecture L finishes at time T. We can show that either L is in C, or there is another optimal C' and L is in C'. Once we have chosen L, we can rule out all lectures that start before T without losing out on any cake.

Coin Changing. Problem: Best way to make change with fewest number of coins: e.g. 34p = 20p + 10p + 2p + 2p. Generally - currency with coins with values $c_1, c_2 ... c_n$. Target value is x and we want to find the smallest number of coins adding up to x. Optimal substructure - 34p = 20p + 10p + 2p + 2p optimal means that 10p + 2p + 2p is optimal for 14p, and 20p + 10p + 2p is optimal for 32p.

Dynamic Programming. M(x) is the smallest number of coins to get target x. Thus, M(x) = min(for k = 1...n) 1 + $M(x - c_k)$. The complexity is O(n)*O(x) = O(nx).

Greedy Algorithm - Cashier's Algorithm. We always start with the largest coin we can. Assuming that the coins are sorted in decreasing size:

Figure 5: Cashier Algorithm



```
for k=1 to n
  m_k = floor(x/c_k) # number of coin k
  x -= m_k*c_k
```

Complexity is O(n)

Leetcode 518: Coin Change 2. You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money. Return the number of combinations that make up that amount. If that amount of money cannot be made up by any combination of coins, return `0`. You may assume that you have an infinite number of each kind of coin. The answer is guaranteed to fit into a signed 32-bit integer.

Code:

```python
def change(self, amount: int, coins: List[int]) -> int:
    # Create a 2D array to store the number of ways to make change for each coin and amount combination
    dp = [[0] * (len(coins)+1) for i in range(amount+1)]

    # Set the first column of the array to 1, since there is only one way to make change for zero cents
    dp[0] = [1] * (len(coins)+1)

    # Loop through each amount from 1 to the target amount
    for a in range(1, amount+1):
        # Loop through each coin from the largest to the smallest
        for i in range(len(coins)-1,-1,-1):
            # Set the current cell in the array to the value of the cell to its right
            dp[a][i] = dp[a][i+1]
            # If the current amount minus the current coin is greater than or equal to zero
            if a-coins[i] >= 0:
                # Add the value of the cell to the left and the cell above it to the current cell
                dp[a][i] += dp[a-coins[i]][i]

    # Return the value in the top-left cell of the array, which represents the number of ways to make change for the target amount using all the coins
    return dp[amount][0]

    # Create a 1D array to store the number of ways to make change for each amount
    dp = [0]*(amount+1)
    # Set the first element of the array to 1, since there is only one way to make change for zero cents
    dp[0] = 1

    # Loop through each coin from the largest to the smallest
    for i in range(len(coins)-1,-1,-1):
        # Create a new array to store the updated values for each amount
        nextDP = [0]*(amount+1)
        # Set the first element of the new array to 1, since there is only one way to make change for zero cents
        nextDP[0]=1

        # Loop through each amount from 1 to the target amount
        for a in range(1, amount+1):
            # Set the current element in the new array to the value of the corresponding element in the previous array
            nextDP[a] = dp[a]
            # If the current amount minus the current coin is greater than or equal to zero
            if a-coins[i] >= 0:
                # Add the value of the current element and the element to its left in the new array to the current element
                nextDP[a] += nextDP[a-coins[i]]
        # Set the previous array to the new array
        dp = nextDP

    # Return the value in the last element of the array, which represents the number of ways to make change for the target amount using all the coins
    return dp[amount]
```

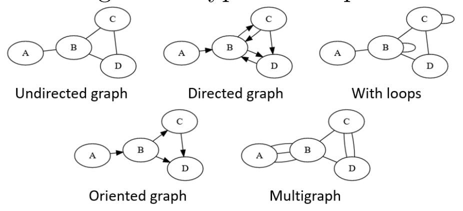
5 Graph Algorithms

5.1 Overview

Definitions. A graph is a collection of vertices/nodes joined by edges/links/-lines. Formally: A pair (V,E) of a set of vertices V and edges E. Each element of E is a pair (u,v) where u ∈ V and v ∈ V. The order ($|V|$) of a graph is equal to the number of vertices, and the size ($|E|$) is equal to the number of edges (but sometimes also $|V| + |E|$). The degree of vertex is equal to the number of edges attached. Indegree = number incoming (in directed graphs) and outdegree = number outgoing. A graph is regular if all vertices have the same degree. A complete graph K_n = n vertices; all possible edges.

More Definitions. A subgraph is a graph formed by a subset of vertices and edges. An induced subgraph is formed by taking a subset of vertices and

Figure 6: Types of Graphs

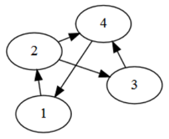


all edges connected to them. Graphs may either be edge-weighted or vertex-weighted. A path is a sequence of vertices with edges between each consecutive pair. A cycle is a path with the same start and end vertex. A graph is acyclic if there are no cycles in it. A path is simple if there are no repeated vertices. A graph is connected if every pair of vertices can be joined by a path. We can divide a graph into its connected components. A tree is a graph where each pair of vertices is connected precisely by one simple path. Trees are always connected and acyclic. A forest is a collection of trees. A bipartite graph is one where the vertices are divided into two subsets U and V. Edges can only be between vertices in U and V.

Graph Data Structures - Adjacency Matrix. Many different ways to represent graphs in code. Adjacency Matrix M: $M_{ij} = 1$ if there is an edge between i and j, else 0. Directed graphs (undirected - symmetric matrix). Loops are allowed. In a multigraph, let M_{ij} be the number of edges from i to j. The storage cost is $\theta(V^2)$.

Figure 7: Adjacency Matrix

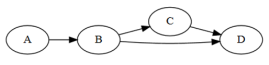
$$M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$



Graph Data Structures - Adjacency Lists and Adjacency Sets. Adjacency List: For each vertex, store list of vertices connected by edge. Storage cost is $\theta(V + E)$. Adjacency Set: Use set instead of list as values. This is more efficient for add/remove/lookup. However, there is more overhead.

Figure 8: Adjacency List

```
G = {'A': ['B'],
     'B': ['C', 'D'],
     'C': ['D'],
     'D': []}
```



Comparison. In a *dense* graph, most vertices have edges. Thus, the adjacency matrix is more efficient (but has the same complexity). In a *sparse* graph, few vertices have edges. Thus, adjacency lists or sets are more efficient.

	Matrix	Lists	Sets
Storage	$O(V^2)$	O(V+E)	O(V+E)
Add Vertex	$O(V^2)$	O(1)	O(1)
Add Edge	O(1)	O(1)	O(1)
Remove Vertex	$O(V^2)$	O(E)	O(V)
Remove Edge	O(1)	O(V)	O(1)
Lookup Edge	O(1)	O(V)	O(1)

5.2 Graph Search

Two main ways. Breadth-first: Start with one vertex. Look at all adjacent vertices. Look at all their adjacent vertices. Depth-first: Start with one vertex. Pick one adjacent vertex. Pick one adjacent vertex of that. Continue until you run out. Backtrack and continue. In both cases, complexity is O(V+E).

BFS Code (educative.io):

```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}

visited = [] # List to keep track of visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)

  while queue:
    s = queue.pop(0)
    print (s, end = " ")

    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
bfs(visited, graph, 'A')
```

DFS Code (educative.io):

```python
# Using a Python dictionary to act as an adjacency list
graph = {
    'A' : ['B','C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F'],
    'F' : []
}

visited = set() # Set to keep track of visited nodes.

def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
dfs(visited, graph, 'A')
```

Predecessor Subgraph. When we do graph search, store a copy of the vertex we arrive from. This is a subgraph, tree and DAG. Breadth-first search gives the shortest path from the start vertext to all others. The complexity is O(V+E).

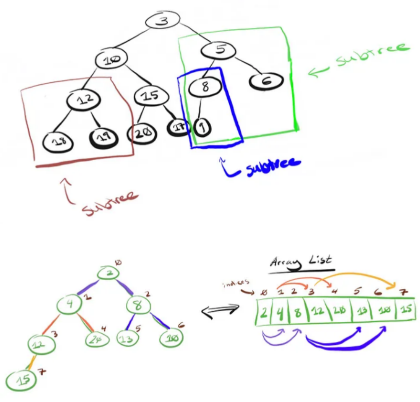
Storage. We will need a data structure that lets us (1) maintain a 'priority' for each item, (2) efficiently extract the highest priority item and (3) efficiently increase the priority of an item. We can implement any priority queue for (1) and (2) - so implementations can include heaps and binary trees. Fibonacci heaps can be used if (3) is important.

Heaps. Heaps are graphs that have the following properties: (1) each node has at most two descending nodes (children nodes), (2) the root node has the smallest or highest value of all the values in the graph and (3) every subtree is itself a heap with the same properties as the overall heap. If the value is the smallest at the root of that subtree, it is a min-heap. If the value was instead the largest, we would call this a max-heap. Heaps are typically represented as array lists.

Binary Heap. A Binary Heap is a complete Binary Tree which is used to store data efficiently to get the max or min element based on its structure. Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(log N) time.

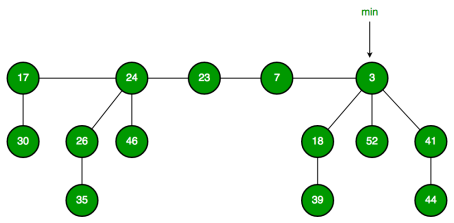
Fibonacci Heap. Fibonacci Heap is a collection of trees with min-heap or

Figure 9: Heap (taken from Try Khov on Medium)



max-heap properties. In Fibonacci Heap, trees can have any shape even if all trees can be single nodes (This is unlike Binomial Heap where every tree has to be a Binomial Tree). Fibonacci Heap maintains a pointer to the minimum value (which is the root of a tree). All tree roots are connected using a circular doubly linked list, so all of them can be accessed using a single 'min' pointer. The main idea is to execute operations in a "lazy" way. For example merge operation simply links two heaps, insert operation simply adds a new tree with a single node. The operation extract minimum is the most complicated operation. It does delay the work of consolidating trees. This makes delete also complicated as delete first decreases the key to minus infinite, then calls extract minimum. Fast amortized running time: The running time of operations such as insert, extract-min and merge in a Fibonacci heap is O(1) for insert, O(log n) for extract-min and O(1) amortized for merge, making it one of the most efficient data structures for these operations.

Figure 10: Fibonacci Heap



	Binary Heap	Fibonacci Heap
Initialise New Heap	O(1)	O(1)
Insert New Item	θ(logn)	O(1)
Find Highest Priority Item	O(1)	O(1)
Find and Remove Highest Priority Item	θ(logn)	O(logn)
Union of Two Heaps	θ(n)	O(1)
Increase Priority of item	θ(logn)	O(1)
Delete Item	θ(logn)	O(logn)

Note: Binary Heap is often more efficient in practice for most n.

5.3 Dijkstra's Algorithm

Overview. Dijkstra's algorithm finds the shortest weighted path using modified breadth-first search, a greedy algorithm. Breadth-first search = always look next at the vertex with the shortest path to the start vertex. Dijkstra's algorithm = same thing but keep track of weights. Dijkstra's Algorithm can only be used when the graph has non-negative edge weights. (1) Store minimum weighted distance to each vertex, (2) Maintain list of possible next vertices to add, (3) At each step, add the one that would make the shortest path, then update the list. Its complexity depends on the data structure used: List - O($V^2 + E$), Fibonacci Heap - O(VlogV + E).

Proof of Correctness. Loop Invariant: Set of shortest distances found correct. Initialisation: Easy, none found yet. Termination: Directly proves what we want. Maintenance: Proof by contradiction, assume counterexample. Take shortest counterexample. Two paths; one found by algorithm, another shorter. Previous node on shorter path has correct distance. When this node was added, it would have put the counterexample node in the candidate list. This would have been used, so contradiction.

Dijkstra's Algorithm Code (Alexey Klochay):

```
import sys

class Graph(object):
    def __init__(self, nodes, init_graph):
        self.nodes = nodes
        self.graph = self.construct_graph(nodes, init_graph)

    def construct_graph(self, nodes, init_graph):
        '''
        This method makes sure that the graph is symmetrical. In
    other words, if there's a path from node A to B with a value V,
     there needs to be a path from node B to node A with a value V.
        '''
        graph = {}
```

```python
        for node in nodes:
            graph[node] = {}

        graph.update(init_graph)

        for node, edges in graph.items():
            for adjacent_node, value in edges.items():
                if graph[adjacent_node].get(node, False) == False:
                    graph[adjacent_node][node] = value

        return graph

    def get_nodes(self):
        "Returns the nodes of the graph."
        return self.nodes

    def get_outgoing_edges(self, node):
        "Returns the neighbors of a node."
        connections = []
        for out_node in self.nodes:
            if self.graph[node].get(out_node, False) != False:
                connections.append(out_node)
        return connections

    def value(self, node1, node2):
        "Returns the value of an edge between two nodes."
        return self.graph[node1][node2]

    def dijkstra_algorithm(graph, start_node):
        unvisited_nodes = list(graph.get_nodes())

        # We'll use this dict to save the cost of visiting each node and update it as we move along the graph
        shortest_path = {}

        # We'll use this dict to save the shortest known path to a node found so far
        previous_nodes = {}

        # We'll use max_value to initialize the "infinity" value of the unvisited nodes
        max_value = sys.maxsize
        for node in unvisited_nodes:
            shortest_path[node] = max_value
        # However, we initialize the starting node's value with 0
        shortest_path[start_node] = 0

        # The algorithm executes until we visit all nodes
        while unvisited_nodes:
            # The code block below finds the node with the lowest score
            current_min_node = None
            for node in unvisited_nodes: # Iterate over the nodes
                if current_min_node == None:
                    current_min_node = node
                elif shortest_path[node] < shortest_path[current_min_node]:
                    current_min_node = node

            # The code block below retrieves the current node’s neighbors and updates their distances
            neighbors = graph.get_outgoing_edges(current_min_node)
            for neighbor in neighbors:
                tentative_value = shortest_path[current_min_node] + graph.value(current_min_node, neighbor)
                if tentative_value < shortest_path[neighbor]:
                    shortest_path[neighbor] = tentative_value
                    # We also update the best path to the current node
                    previous_nodes[neighbor] = current_min_node

            # After visiting its neighbors, we mark the node as "visited"
            unvisited_nodes.remove(current_min_node)

        return previous_nodes, shortest_path
```

Leetcode 743: Network Delay Time. You are given a network of n nodes, labelled from 1 to n. You are also given times, a list of travel times as directed edges times[i] $= (u_i, v_i, w_i)$ where u_i is the source node, v_i is the target node, and w_i is the time it takes for a signal to travel from source to target. We will send a signal from a given node k. Return the time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

Code:

```python
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        edges = collections.defaultdict(list)
        for u, v, w in times:
            edges[u].append((v, w))
        minHeap = [(0,k)]
        visit = set()
        t = 0

        while minHeap:
            w1, n1 = heapq.heappop(minHeap)
            if n1 in visit:
                continue
            visit.add(n1)
            t = max(t, w1)

            for n2, w2 in edges[n1]:
                if n2 not in visit:
                    heapq.heappush(minHeap, (w1+w2, n2))

        return t if len(visit) == n else -1
```

5.4 Topological Sort

Overview. We have a list of tasks and a graph of dependencies. E.g. how to get dressed. We can use a DFS. Pick a task at random. If it has dependencies, pick one. We repeat until we have a task with no dependencies. This can be our first task. We remove this task from the list and backtrack until we have a new task with no dependencies. This is our second task. We repeat until there are no more tasks.

Topological Sort Code:

```python
""" This code implements a topological sort algorithm, which is
    used to sort the nodes of a directed acyclic graph (DAG) in a
    linear order such that for every directed edge (u, v), node u
    comes before node v in the order.

The topsort function takes in a graph G represented as an adjacency
     list and returns a list order containing the nodes of the
    graph in topologically sorted order. It does this by iterating
    over all nodes in G, and for each node that has not already
    been visited (done), it calls the topsort_recursive function on
     that node.

The topsort_recursive function performs a depth-first search (DFS)
    starting from the given start node. It recursively visits all
    nodes reachable from start in the graph and adds them to the
    order list in reverse order of their finishing time (i.e., the
    last node visited is added first, and so on). The done set is
    used to keep track of nodes that have already been visited to
    avoid revisiting them. """

def topsort(G):
    done = set()
    order = []
    for v in G:
        if v in done:
            continue
        topsort_recursive(G, v, done, order)
    return order

def topsort_recursive(G, start, done, order):
    for v in G[start]:
        if v in done:
            continue
        topsort_recursive(G, v, done, order)
    order.append(start)
    done.add(start)
```

6 Further Topics

6.1 Amortised Analysis

Introduction. Amortised Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. We analyze a sequence of operations and guarantee a worst-case average time that is lower than the worst-case time of a particularly expensive operation.

Dynamic Array Problem. The dynamic array data structure (C++ std::vector, Python List) can grow or shrink dynamically as elements are added or removed. The cost of growing the array is proportional to the size of the array, which can be expensive. However, if we amortise the cost of growing the array over several insertions, the average cost of each insertion becomes constant or less.

We double the reserved size when we run out of space. This is O(n) but only happens 1/n of the time. So in total if append is called n times it will be O(n), but amortised it is O(1).

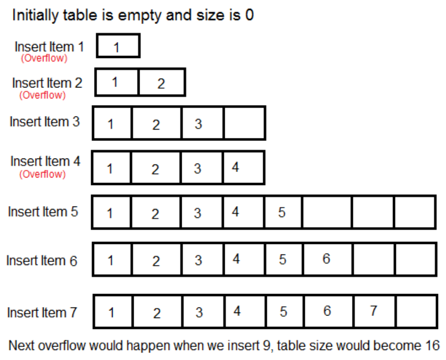
	Linked List	Array	Dynamic Array
`X[i]`	O(n)	O(1)	O(1)
`X.append(item)`	O(1)	O(n)	Amortised O(1), Worst Case O(n)

Hash Table Insertion. How do we decide on table size? There is a trade-off between space and time. If we make the hash-table size big, the search time becomes low, but the space required becomes high. The solution to this is to use Dynamic Tables (aka Arrays). The idea is to increase the size of the table whenever it becomes full. We (1) allocate memory for larger table sizes (typically twice the old table), (2) copy the contents of the old table to a new table, and (3) free the old table. If the table has space available, we simply insert a new item in the available space. Time complexity: O(n) for insertion (worse-case cost). The worse-cast cost of n inserts is thus $O(n^2)$. This gives an upper bound but not a *tight upper bound* for n insertions as all insertions don't take $\theta(n)$ time.

Amortised Analysis of Dynamic Table. We can prove that insertion is O(1) using Amortised Analysis - great for hashing. The method below is called the *Aggregate Method*.

Item number	1	2	3	4	5	6	7	8	9	10
Table Size	1	2	4	4	8	8	8	8	16	16
Cost	1	2	3	1	5	1	1	1	9	1

Figure 11: Dynamic Table



Amortised Cost $= \frac{1+2+3+5+1+1+9+1}{n} \rightarrow$ We can simplify the above series by breaking terms 2,3,5,9 into two as 1+1, 1+2, 1+4, 1+8. Thus this gives Amortised Cost $= \frac{[(1+1+1+1...)+(1+2+4+...)]}{n} \leq \frac{[n+2n]}{n} \leq 3$. So the amortised cost is O(1).

6.2 Multithreaded Analysis of Algorithms

Author's Notes. I used ICS311 from the University of Hawaii to augment Goodman's notes on Multithreading. Check out CLRS #27 for more details. Also, this is probably longer than what you need for the exam.

Introduction. Parallel Computers are cheaper and more ubiquitous - we have supercomputers with custom architectures and networks, compute clusters with dedicated networks (distributed memory), multi-core integrated circuit chips with shared memory, and GPUs (Graphic Processing Units). *Static Threading* is the abstraction of virtual processors. But rather than managing threads explicitly, *Dynamic Multithreading* specifies opportunities for parallelism and a *concurrency platform* manages the decisions of mapping these to static threads (load balancing, communication etc.).

Concurrency Constructs. These keywords reflect current parallel-computing practice. *Parallel* means to add to a loop construct such as for loops to indicate each iteration can be executed in parallel. *Spawn* means to create a parallel subprocess, then keep executing both the newly created subprocess and the current process concurrently (parallel procedure call). *Sync* means to wait here until all active parallel threads created by this instance of the program finish. These keywords specify opportunities for parallelism without affecting whether the corresponding sequential program obtained by removing them is correct.

Parallel Fibonacci. Here is a recursive non-parallel algorithm for computing Fibonacci numbers:

Figure 12: Non-Parallel Fibonacci

```
FIB(n)
if n ≤ 1
    return n
else x = FIB(n − 1)
    y = FIB(n − 2)
    return x + y
```

`Fib` has a recurrence relation T(n) = T(n-1) + T(n-2) + $\theta(1)$, which has solution T(n) = $\theta(F_n) = \theta(\frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}})$. This grows exponentially in n, so it's not very efficient. We note that recursive calls can operate independently of each other, so we compute the two recursive calls in parallel. This illustrates the concurrency keywords.

Figure 13: Parallel Fibonacci

```
P-FIB(n)
if n ≤ 1
    return n
else x = spawn P-FIB(n − 1)
    y = P-FIB(n − 2)
    sync
    return x + y
```

Logical Parallelism: the *spawn* keyword does not force parallelism, it just says that it is permissible. A scheduler will make the decision concerning allocation to processors. However, if parallelism is used, *sync* must be respected. For safety, there is an implicit *sync* at the end of every procedure.

Modelling Dynamic Multithreading. We model multithreaded computation as a computation DAG. Vertices in V are instructions, or strands; sequences of non-parallel instructions. Edges in E represent dependencies between instructions or strands; (u,v) $\in$ E means u must execute before v.

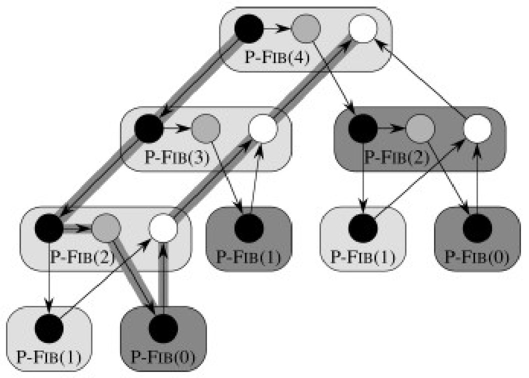
1. Continuation Edges (u,v) are drawn horizontally and indicate that v is the successor to u in the sequential procedure.

2. Call Edges (u,v) point downwards, indicating that u called v as a normal subprocedure call.

3. Spawn Edges (u,v) point downwards, indicating that u spawned v in parallel.
4. Return Edges point upwards to indicate the next strand executed after returning from a normal procedure call, or after parallel spawning at a sync point.

A strand with multiple successors means all but one of them have spawned. A strand with multiple predecessors means they join at a sync statement. If G has a directed path from u to v they are logically in series; else they are logically parallel. We assume an ideal parallel computer with sequentially consistent memory, meaning it behaves as if instructions were executed sequentially in some full ordering consistent with orderings within each thread (i.e. consistent with partial ordering of the computation DAG).

Figure 14: Parallel Fibonacci DAG for P-Fib(4)



Performance Measures. We write T_p to indicate the running time of an algorithm on P processors. Then, we define these measures and laws:

1. Work: T_1 is the total time to execute an algorithm on one processor. This is called work (analogous to physics; the total amount of computational work that gets done). An ideal parallel computer with P processors can do at most P units of work in one time step. So, in T_p time it can do at most $P \cdot T_p$ work. Since the total work is T_1, $P \cdot T_p \geq T_1$, or dividing by P we get the *work law*: $T_p \geq T_1/P$. The work law can be read as saying that the speedup for P processors can be no better than the time with one processor divided by P. *In short, the cost $P \cdot T_p$ is always at least the work T_1.*

2. Span: T_∞ is the total time to execute an algorithm on an infinite number of processors (or more practically speaking, on just as many processors as are needed to allow parallelism wherever it is possible). T_∞ is called the span because it corresponds to the longest time to execute the strands along any path in the computation DAG (the biggest computational span across the DAG). It is the fastest we can possibly expect (Ω bound) because no matter how many processors we have, the algorithm must take this long. Hence the *Span Law* states that a P-processor ideal parallel computer cannot run faster than one with an infinite number of processors: $T_p \geq T_\infty$. This is because at some point the span will limit the speedup possible, no matter how many processors you add. *In short, a finite number of p processors cannot outperform an infinite number of processors.*

3. Speedup: The ratio T_1/T_p defines how much speedup you get with P processors as compared to one. By the work law, one cannot have any more speedup than the number of processors. When the speedup $T_1/T_p = \theta(P)$ we have *linear speedup.* When $T_1/T_p = P$ we have *perfect linear speedup.*

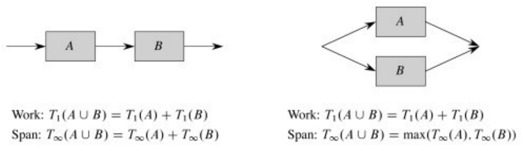
4. Parallelism: The ratio T_1/T_∞ of the work to the span gives the potential parallelism of the computation. It can be interpreted three ways: (1) Ratio: the amount of work that can be performed for each step of parallel execution time. (2) Upper Bound: The maximum possible speedup that can be achieved on any number of processors. (3) Limit: The limit on the possibility of attaining perfect linear speedup. Once the number of processors exceeds the parallelism, the computation cannot achieve perfect linear speedup. The more processors we use beyond parallelism, the less perfect the speedup.

Parallel Slackness. $(T_1/T_\infty)/P = T_1/(P \cdot T_\infty)$, Parallel Slackness, is the factor by which the parallelism of the computation exceeds the number of processors in the machine. If the slackness is less than 1, then perfect linear speedup is not possible; you have more processors than you can make use of. If slackness is greater than 1, then the work per processor is the limiting constraint and a scheduler can strive for linear speedup by distributing the work across more processors.

Analysing Work. Simple - ignore the parallel constructs and analyse the serial algorithm. For example, the work of P-Fib(n) is $T_1(n) = T(n) = \theta(F_n)$.

Analysing Span. If in series, the span is the sum of the spans of the subcomputations. This is similar to normal sequential analysis. If in parallel, the span is the maximum of the spans of the subcomputations.

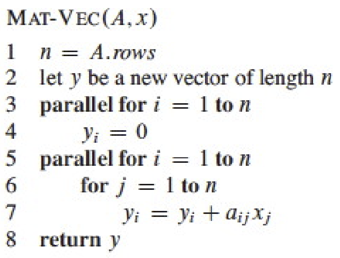
Figure 15: Analysing Span


A
B
Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$
A
B
Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

The span of the parallel recursive calls of P-Fib(n) is: $T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) = T_\infty(n-1) + \Theta(1)$, which has solution $\theta(n)$. The parallelism of P-Fib(n) in general is $\frac{T_1(n)}{T_\infty} = \Theta\left(\frac{F_n}{n}\right)$, which grows dramatically as F_n grows much faster than n. There is considerable parallel slackness, so above small n, there is potential for near perfect linear speedup.

Parallel Loops. The parallel keyword is used with loop constructs such as for. Suppose we want to multiply an n x n matrix A = (a_{ij}) by an n-vector x = (x_j). This yields an n-vector y = (y_i) where: $y_i = \sum_{j=1}^{n} a_{ij}x_j$ for i = 1,2...n. The following algorithm does this in parallel.

Figure 16: Multiply Vector Parallel



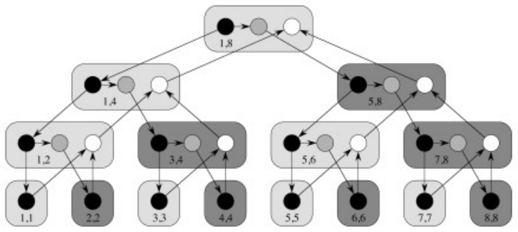
```
MAT-VEC(A, x)
n = A.rows
let y be a new vector of length n
parallel for i = 1 to n
    $y_i = 0$
parallel for i = 1 to n
    for j = 1 to n
        $y_i = y_i + a_{ij} x_j$
return y
```

The `parallel for` keywords indicate that each iteration of the loop can be executed concurrently. Notice that the inner for loop is not parallel; a possible point of improvement to be discussed.

Implementing Parallel Loops. It is not realistic to think that all n subcomputations in these loops can be spawned immediately with no extra work (for *some* operations on *some* hardware up to a constant n this may be possible, such as hardware designed for matrix operations, but we are concerned with the general case). This can be accomplished by a compiler with a divide and conquer approach, itself implemented with parallelism. The procedure shown below is called with MAT-VEC-MAIN-LOOP (A,x,y,n,1,n). Lines 2 and 3 are the lines originally within the loop.

Figure 17: Implementing Parallel Loops

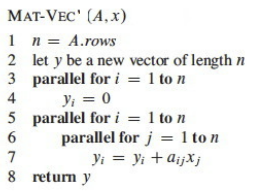
```
MAT-VEC-MAIN-LOOP$(A, x, y, n, i, i')$
1  if $i == i'$
2      for $j = 1$ to $n$
3          $y_i = y_i + a_{ij} x_j$
4  else $mid = \lfloor (i + i')/2 \rfloor$
5      spawn MAT-VEC-MAIN-LOOP$(A, x, y, n, i, mid)$
6      MAT-VEC-MAIN-LOOP$(A, x, y, n, mid + 1, i')$
7      sync
```



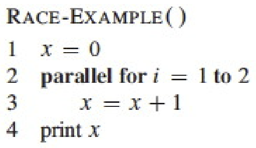
The computation DAG is shown. It appears that a lot of work is done to spawn the n leaf node computations, but the increase is not asymptotic. The work of MAT-VEC is $T_1(n) = \theta(n^2)$ due to nested loops in 5-7. Since the tree is a full binary tree, the number of internal nodes is 1 fewer than leaf nodes, so this extra work is also $\theta(n)$. So the work of recursive spawning contributes a constant factor when amortised across the work of the iterations. However, concurrency platforms sometimes coarsen the recursion tree by executing several iterations in each leaf, reducing the amount of recursive spawning. The span is increased by $\theta(logn)$ due to the tree. In some cases, such as this one, this increase is washed out by other dominating factors such as doubly nested loops.

Nested Parallelism and Race Conditions. The span is $\theta(n)$ because even with full utilisation of parallelism, the inner for loop still requires $\theta(n)$. Since the work is $\theta(n^2)$, the parallelism is $\theta(n)$. We can try making the inner for loop parallel as well, but this leads to a *determinancy race*, when we outcome of a computation is nondeterministic. This happens when two logically parallel computations access the same memory and one performs a write. For example, in RACE, the output might be 1 or 2 depending on the order in which access to x is interleaved by two threads.

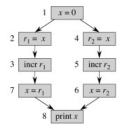
Figure 18: MAT-VEC' and Race Example

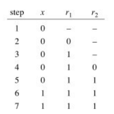


```
MAT-VEC' (A, x)
n = A.rows
let y be a new vector of length n
parallel for i = 1 to n
    y_i = 0
parallel for i = 1 to n
    parallel for j = 1 to n
        y_i = y_i + a_ij x_j
return y
```



```
RACE-EXAMPLE()
x = 0
parallel for i = 1 to 2
    x = x + 1
print x
```

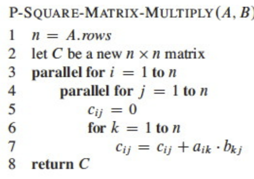




step	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

Parallelised Matrix Multiplication. Here is an algorithm for parallel matrix multiplication, based on the $T_1(n) = \theta(n^3)$ algorithm.

Figure 19: Parallelised Matrix Multiplication

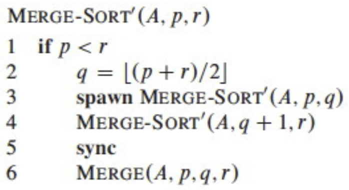


```
P-SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
parallel for i = 1 to n
    parallel for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

The span of this algorithm is $T_\infty(n) = \theta(n)$ due to the path for spawning the outer and inner parallel loop executions and then the n executions of the innermost for loop. So the parallelism is $\frac{T_1(n)}{T_\infty(n)} = \frac{\Theta(n^3)}{\Theta(n)} = \Theta(n^2)$.

Parallelised Merge Sort. The dividing is in the main procedure MERGE-SORT, so we can parallelise it by spawning the first recursive call:

Figure 20: Parallelised Merge Sort



```
MERGE-SORT'(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    spawn MERGE-SORT'(A, p, q)
    MERGE-SORT'(A, q + 1, r)
    sync
    MERGE(A, p, q, r)
```

MERGE remains a serial algorithm, so its work and span are $\theta(n)$ as before. The recurrence for the work $MS'_1(n)$ of MERGE-SORT' is the same as the serial version: $MS'_1(n) = 2MS'_1(n/2) + \theta(n) = \theta(nlogn)$.

The recurrence for the span $MS'_\infty(n)$ of MERGE-SORT' is based on the fact that the recursive calls run in parallel, so there is only one n/2 term: $MS'_\infty(n) = MS'_\infty(n/2) + \theta(n) = \theta(n)$. The parallelism is thus $MS'_1(n)/MS'_\infty(n) = \theta(nlgn/n) =$

$\theta(lgn)$. This is low parallelism, meaning that even for large input we would not benefit from having hundreds of processors. One way to address is is to speed up the serial MERGE.

Parallel Sum. Assess the work, span and parallelism of `psum`. The work $T_1(n) = \theta(n)$. The span $T_\infty(n) = \theta(1) + T_\infty(n/2) = \theta(logn)$. The parallelism $T_1/T_\infty = \theta(n/logn)$.

Figure 21: Parallelised Sum

```
def psum(X):
    n = len(X)
    if n==1 return X[1]
    if n==0 return 0
    spawn L = psum(X[1 to n/2])
    spawn R = psum(X[n/2+1 to n])
    sync
    return L+R
```

6.3 Greedy Scheduler

Introduction. So far, we assumed a perfect scheduler. It is relatively easy to get close to this (2x at worst). In a Greedy Scheduler, we assign any requested strand to any free processor. The performance is $T_p \leq \frac{T_1}{P} + T_\infty$.

Proof: T_p = T(All Processors Busy) + T(Some Processors Free). The total when all processors busy = PT(All busy) $\leq T_1$ = Total Work. So T(all busy) $\leq T_1/P$ When some processors free, its as if P = ∞. So T(Some Free) $\leq T_\infty$

Performance is close to ideal time T_p^*. According to the Work Law, $T_p^* \geq \frac{T_1}{P}$, and Span Law, $T_p^* \geq T_\infty$. So, $T_p^* \geq max(\frac{T_1}{P}, T_\infty)$. The greedy scheduler gives $T_p \leq \frac{T_1}{P} + T_\infty \leq 2\cdot$ max $(T_1/P, T_\infty) \leq 2T_p^*$. The performance is not worse than twice as bad as ideal; the speedup is also close to perfect linear, if parallelism is high.

If $P << T_1/T_\infty$ then $T_P \approx T_1/P$ so speedup $\approx$ P. $P << T_1/T_\infty$ means $T_\infty << T_1/P$. The greedy scheduler gives $T_p \leq \frac{T_1}{P} + T_\infty \approx \frac{T_1}{P}$. According to the Work Law, $T_P \geq T_1/P$ so $T_P \approx \frac{T_1}{P}$.

6.4 Intractability

Definition. Problem A polynomially reduces to problem B if (1) it can transform instance of A into instance of B in polynomial time, (2) it can transform solution of B into solution of A in polynomial time. We write $A \leq_p B$. If you can solve B in polynomial time, then we can solve A in polynomial time too.

P and NP. P problems represent those solvable in polynomial time. NP problems represent those verifiable in polynomial time. Problem Q is NP-hard if for all NP problems R, we can polynomially reduce R to Q, i.e. $R \leq_P Q$. NP-Complete = NP and NP-hard.

Finding NP-Complete Problem Q. It is easy to show that it is NP (we can give an algorithm to check solution). It is hard to show every NP problem R can be polynomially reduced to Q. If R is NP-Complete and $R \leq_P Q$ then Q is NP-hard. If Q is also NP then Q is NP-complete. If R is NP-complete then for all NP S, $S \leq_P R$. We know that $R \leq_P Q$ so $S \leq_P Q$. So Q is NP-hard.

6.5 Halting Problem

Overview. Suppose we have a program `halts(f)` that returns whether or not the program `f` halts or not.

```
def g() if halts (g) then loop forever
```

Does `g` halt, if yes, then it must loop forever. If no, then it must halt. So the `halts` program cannot exist. We need to do some work to fix the self-reference.

Fixing the self-reference. H(i,x) is the mathematical function that returns 1 if program numbered i halts on input x, otherwise 0. Suppose there were a program `h` that implements it.

```
def g(i) if h(i,i) == 0 then return 0 loop forever
```

Can `h(e,e)` = H(e,e)? 1. `h(e,e)` = 0 therefore g(e) halts thus H(e,e) = 1. 2. `h(e,e)` = 1 therefore g(e) loops forever thus H(e,e) = 0. So h cannot implement H.

7 Useful Resources

1. MIT 6.046J Design and Analysis of Algorithms
2. Neetcode.io
3. Introduction to Algorithms, or "CLRS"