CSC373 Notes

Divide and Conquer: A general algorithmic framework involving dividing problems into subproblems, solving them recursively & independently, then combining them to solve what remains.

- Proof of correctness is usually simple, but running-time can be more complicated

Master Function: Let $a \geq 1, b > 1, f(n)$ be a function, $T(n) \leq a \cdot T(\frac{n}{b}) + f(n)$ (where $\frac{n}{b}$ can also be $\lceil \frac{n}{b} \rceil$). Then,

$$T(n) = \begin{cases} \mathcal{O}(n^{\log_b a}) & \text{if } f(n) = \mathcal{O}(n^{\log_b a - \epsilon}) \text{ for some } \epsilon > 0 \\ \mathcal{O}(n^{\log_b a} \log^{k+1} n) & \text{if } f(n) = \mathcal{O}(n^{\log_b a} \log^k n) \text{ for some } k \geq 0 \\ \mathcal{O}(f(n)) & \text{if } f(n) = \mathcal{O}(n^{\log_b a + \epsilon}) \text{ for some } \epsilon > 0 \end{cases}$$

eg. Counting **inversions** – index pairs (i, j) of list L such that $i < j, L[i] > L[j]$

Suppose the split of L is two sorted lists:

$$A = [1,2,6,8,10], B = [2,3,5,7,12]$$

```
Initialize $i = j = 0$
If $A[i] < B[j]$,
    Set $i \leftarrow i + 1$
If $A[i] > B[j]$,
    Then $(i, j), (i, j-1), \dots, (i, 0)$ are inversions!
    Set $j \leftarrow i + 1$
```

Notice the iterations resembles the Merge helper function from MergeSort!

The resulting function will count inversions and sort!

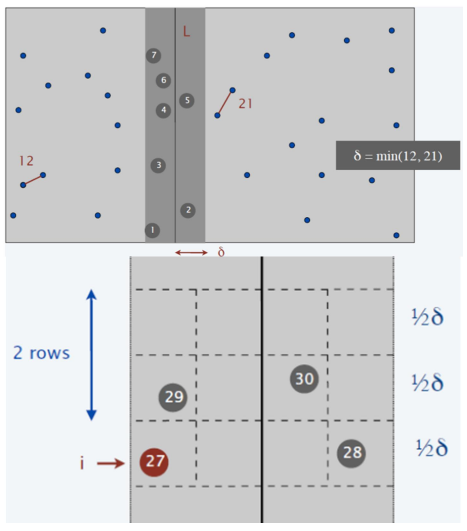
```
def count_inversions(L):
    if len(L) == 1:
        return 0, L
    A, B = L[:len(L) // 2], L[len(L) // 2:]
    a, A = count_inversions(A)
    b, B = count_inversions(B)
    c, L' = merge(A, B) # Modified as shown on left
    return a + b + c, L'
```

Merge is $\mathcal{O}(n)$, so the running time is

$$T(n) \leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n) \Rightarrow \mathcal{O}(n \log n)$$

eg. Given n points in 2D, find the **closest pair of points**

Assume no points share an x or y coordinate – this simplifies some technicalities.



Algorithm:

1) Divide space vertically at x to contain equal points
2) Recursively find min distances, δ_1, δ_2. Let $\delta = \min\{\delta_1, \delta_2\}$
3) Sort points in the range $[x - \delta, x + \delta]$ by y-value
4) Check each point's distance with the list's next 11 points
5) Return the minimum of the min distance and δ.

Why check the next 11 points?

- Split $[x - \delta, x + \delta]$ into grids of $\frac{\delta}{2} \times \frac{\delta}{2}$
- Points in $[x - \delta, x]$ are $\geq \delta$ apart from each other
- Points in $[x, x + \delta]$ are $\geq \delta$ apart from each other
- Therefore, no points lie in the same grid
- Point i is in any 4 grids of the bottom row *(see left)*
- Worst case – the next 11 points in the list are in the next 11 grids. Thus, check 11 next points.

The running-time is

$$T(n) \leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n \log n) \Rightarrow \mathcal{O}(n \log^2 n)$$

But can be improved to $\mathcal{O}(n \log n)$ by globally sorting on y-value at the beginning and referencing that order.

eg. **Karatsuba's algorithm** for faster multiplication of n-digit integers x and y

Let $x = 10^{\frac{n}{2}} x_1 + x_2$, where x_1, x_2 are the first/last $\frac{n}{2}$ digits of x
Let $y = 10^{\frac{n}{2}} y_1 + y_2$, where y_1, y_2 are the first/last $\frac{n}{2}$ digits of y

$$\begin{aligned} xy &= 10^n x_1 y_1 + 10^{\frac{n}{2}}(x_1 y_2 + x_2 y_1) + x_2 y_2 \\ &= 10^n x_1 y_1 + 10^{\frac{n}{2}}\big((x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2\big) + x_2 y_2 \end{aligned}$$

Instead of 4 products $x_1 y_1, x_2 y_2, x_2 y_1, x_1 y_2$
Calculate 3 products $x_1 y_1, x_2 y_2, (x_1 + x_2)(y_1 + y_2)$
Don't worry about the 10^n and $10^{\frac{n}{2}}$ – on a computer these operations don't matter.
There are extra additions, but multiplications are much more costly in hardware, so this saves time!

$$T(n) \leq 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \Rightarrow \mathcal{O}(n^{\log_2 3})$$

Strassen's algorithm is a $\mathcal{O}(n^{\log_7 8})$ way to multiply two $n \times n$ matrices by recursively partitioning the matrices into 2×2 block matrices and similarly performing additions to avoid multiplications.

eg. **Selection** – finding the k-th smallest element of a list L

QuickSelect Algorithm:

1) Pick a pivot p *(see below for how)*
2) Divide L into $L_{\leq p}, L_{>p}$
3) If $|L_{\leq p}| \geq k$, run QuickSelect$(L_{\leq p}, k)$
4) If $|L_{\leq p}| < k$, run QuickSelect$(L_{>p}, k - |L_{\leq p}|)$

Pivot Choosing

1) Divide L into $\frac{n}{5}$ groups of 5
2) Find the median of each group directly
3) Use the median of the $\frac{n}{5}$ medians as pivot $\left(= T(\frac{n}{5})\right)$

For pivot choosing…

- $\frac{1}{2} \cdot \frac{n}{5} = \frac{n}{10}$ medians are $\leq p$.
- For those medians, ≥ 3 of the 5 median-ed values are $\leq$ the median
- $\geq 3 \times \frac{n}{10} = \frac{3}{10} n$ elements in L are $\leq p$
- Thus, $\leq \frac{7}{10} n$ elements in L are $\geq p$

Thus, for QuickSelect,

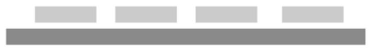
$$T(n) \leq \underbrace{T\left(\frac{n}{5}\right)}_{\text{medians}} + \underbrace{T\left(\frac{7}{10} n\right)}_{\text{quickselect}} + \mathcal{O}(n) \Rightarrow \mathcal{O}(n)$$

Greedy Algorithm: Algorithms that make the locally most optimal decision each time to solve a problem

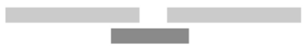
- Most run trivially in polynomial time, but require proofs of optimal solutions

eg. **Interval scheduling** – finding a maximally-sized subset of non-conflicting intervals

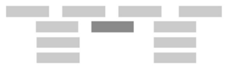
Our algorithm picks intervals one-by-one based on some metric – here are some that don't work:



Earliest Start Time



Shortest Interval



Fewest Conflicts

The correct metric is earliest finish time – this produces a $\mathcal{O}(n \log n)$ algorithm

1) Sort intervals by finish time
2) For each interval, choose it if its start time is after the last chosen job's finish time (ie. $(i+1)_{\text{start}} \geq i_{\text{finish}}$)

Proof of optimality (via contradiction).
Suppose the greedy solution (GS), $i_1, \dots, i_n$ *(sorted by finish time)*, is not optimal.
Consider an optimal solution (OS) $I_1, \dots, I_m$ *(sorted by finish time)* where $I_1 = i_1, \dots, I_k = i_k$ for the largest k.
We claim $k < n \leq m$:

- $n \leq m$ holds as otherwise the GS is longer than the OS
- $k < n$ holds, as otherwise,
  - If $k > n$, this is impossible as by definition, $k = \min\{n, m\}$, so $k \leq n$ and $k \leq m$
  - If $k = n$, then there exists an extra interval i_{n+1} chosen by the OS but not the GS. But since $i_n = I_n$ and i_{n+1} fits in the OS, it must also fit into the GS, so it would be chosen, a contradiction

Consider solution $i_1, \dots, i_k, i_{k+1}, I_{k+2} \dots, I_m$

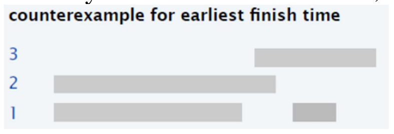
- Replacing I_{k+1} with i_{k+1} is fine as $(i_{k+1})_{\text{finish}} \leq (I_{k+1})_{\text{finish}}$; otherwise, GS would've picked I_{k+1}
- This solution is optimal (has m intervals), but matches the GS in $k+1$ indices, a contradiction!

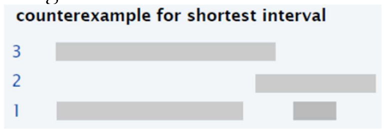
Proof of optimality (via induction).
Define S_k as the GS's solution so far after considering the first k intervals by increasing finish time, so $S_0 = \emptyset$
Define a partial solution as **promising** if it is a subset of an OS
We claim $\forall k \in \{0, \dots, n\}, S_k$ is promising
Base Claim: $S_0 = \emptyset$ is a subset of any set, so it is a subset of an OS.
Inductive Step: Assume S_k is promising, show S_{k+1} is promising

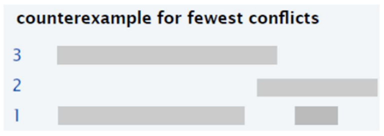
- If the GS does not select i_{k+1}:
    - i_{k+1} conflicts with something in S_i
    - Since S_{k+1} is a subset of an OS, that OS cannot include i_{k+1}
    - Thus, $S_{k+1} = S_k$ which is promising
- If the GS selects i_{k+1}:
    - Thus $S_{k+1} = S_k \cup \{i_{k+1}\}$
    - Consider the earliest interval I_{k+1} in the OS and not in S_k; replace it with i_{k+1}
    - The solution will still be feasible as $(i_{k+1})_{\text{finish}} \leq (I_{k+1})_{\text{finish}}$, and still is optimal
    - Thus S_{k+1} is a subset of that OS, and is promising

eg. **Interval partitioning** –scheduling lectures (intervals) for as little classrooms (slots) as possible

Greedily choose based on a metric, creating a new classroom whenever there's a conflict.







The correct metric is earliest start time – this produces a $\mathcal{O}(n \log n)$ algorithm

1) Sort intervals by finish time
2) For each interval, if it is compatible with a slot, schedule it there; otherwise, allocate a new slot for it
3) Return the schedule

To make sure the time is $\mathcal{O}(n \log n)$,

- Slots are stored in a **priority queue** (min heap), where key = latest finish time of all intervals in it
- To check an interval's compatibility, compare its start time with the minimum key
    - If compatible, add the interval to that slot and increase the key
    - If not compatible, create new slot, add interval to slot, set the key to the finish time
- Check all intervals + heap operations = $\mathcal{O}(n \log k)$, where k = number of slots
- Since $k \leq n$, total time is still $\mathcal{O}(n \log n)$

Proof of optimality (bounds).
Let k = number of slots needed optimally
Let d = maximum number of intervals at same time
Let g = number of slots used by GS
Then $d \leq k$ holds; otherwise, if $d > k$, there are $> k$ simultaneous intervals with not enough slots
Then $g \leq d$ holds

- Slot g opened because previous slots have ≥ 1 conflicting interval I_i with new interval i_g
- Consider the intervals $I_1, \dots, I_{g-1}, i_g$
- Since we sorted by start time, all I_i the GS previously chose start before $(i_g)_{\text{start}}$ and end after $(i_g)_{\text{start}}$
- At time $(i_g)_{\text{start}}$, there're $\geq g - 1$ other lectures in the middle of running
- So, there're $\geq g$ conflicting lectures, thus $d \geq g$

Thus $g \leq k$. And since by definition, $k \geq g$, then $g = k$.

Note: the interval problems can be thought of as graph problems:

- Interval scheduling = maximal independent set (ie. set of vertices where none are connected)
- Interval partitioning = graph colouring

eg. Minimizing **lateness** – a machine does jobs j taking j_{time} time due at j_{due}. Jobs can be scheduled at j_{start} to finish at $j_{\text{finish}} = j_{\text{start}} + j_{\text{time}}$. Define lateness as $j_{\text{late}} = \max\{0, j_{\text{finish}} - j_{\text{due}}\}$. Minimize $\max_{i\in\{1,\ldots,n\}}(j_i)_{\text{late}}$.

Greedily choose jobs based on a metric. Here are some that don't work:

$j = 1$: $1_{\text{time}} = 1$, $1_{\text{due}} = 100$
$j = 2$: $2_{\text{time}} = 10$, $2_{\text{due}} = 10$
Shortest Time

$j = 1$: $1_{\text{time}} = 1$, $1_{\text{due}} = 2$
$j = 2$: $2_{\text{time}} = 10$, $2_{\text{due}} = 10$
Shortest Slack, $i_{\text{due}} - i_{\text{time}}$

The correct metric is <u>earliest deadline</u> – this produces a $\mathcal{O}(n \log n)$ algorithm

1) Sort jobs by deadline, set $t \leftarrow 0$
2) Iteratively set $(j_i)_{\text{start}} \leftarrow t$, then $(j_i)_{\text{finish}} \leftarrow t + (j_i)_{\text{time}}$, and $t \leftarrow t + (j_i)_{\text{time}}$
3) Return $\left[[(j_i)_{\text{start}}, (j_i)_{\text{finish}}], \ldots\right]$

Proof of optimality (via contradiction).
Consider the following lemmas:

- There exists an OS with no idle time
- GS has no idle time
- Only GS has no inversions – pairs of jobs (j_1, j_2) where $(j_1)_{\text{due}} > (j_2)_{\text{due}}$ and $(j_1)_{\text{start}} < (j_2)_{\text{start}}$
- If a schedule with no idle time has an inversion, it has an inversion in an adjacent pair of jobs
- Swapping adjacent inversions doesn't increase lateness, reduces inversions by 1
    - Let (j_1, j_2) be the adjacent inversion so $(j_1)_{\text{due}} > (j_2)_{\text{due}}$
    - Since j_2 is being flipped earlier, $(j_2')_{\text{late}} \leq (j_2)_{\text{late}}$
    - Let i' be job i after the swap, so $(j_2')_{\text{late}} \leq (j_2)_{\text{late}}$

$$\begin{aligned}(j_1')_{\text{late}} &= \max\{0, (j_1')_{\text{finish}} - (j_1)_{\text{due}}\}\\ &= \max\{0, (j_2)_{\text{finish}} - (j_1)_{\text{due}}\}\\ &\leq \max\{0, (j_2)_{\text{finish}} - (j_2)_{\text{due}}\}\\ &= (j_2)_{\text{late}}\end{aligned}$$

    - Since $(j_1')_{\text{late}} \leq (j_2)_{\text{late}}$ and $(j_2')_{\text{late}} \leq (j_2)_{\text{late}}$, then $\max_{i\in\{1,\ldots,n\}}(j_i')_{\text{late}} \leq \max_{i\in\{1,\ldots,n\}}(j_i)_{\text{late}}$
    - And trivially, the inversion is no longer an inversion after the flip.

Suppose GS is not optimal
Consider an OS with the fewest inversions and no idle time.
Because GS is not optimal, OS has an inversion, and thus an adjacent inversion
But then swapping the pair keeps the optimality and reduces the inversion.
So, swapping all adjacent inversions preserves optimality, but this is also the GS. Contradiction!

Proof of optimality (via induction).
We claim there exist an OS with $\leq k$ inversions for any $k \in \left\{0, \ldots, \binom{n}{2}\right\}$
<u>Base Claim:</u> For $k = \binom{n}{2}$, any OS has at most $\binom{n}{2}$ inversions so it'll work.
<u>Inductive Step:</u> Assume there's an OS with $\leq k$ inversions, show there's an OS with $\leq k - 1$ inversions

- If the OS has $\leq k - 1$ inversions, we're done
- If the OS has k inversions, there's a version of that OS with no idle time. Find an adjacent inverted pair and swap it, preserving optimality and reducing inversions to $k - 1$.

eg. Lossless compression – represent text with n unique symbols in $\log_2 n$ bits; so texts of length m has $m \log_2 n$ bits

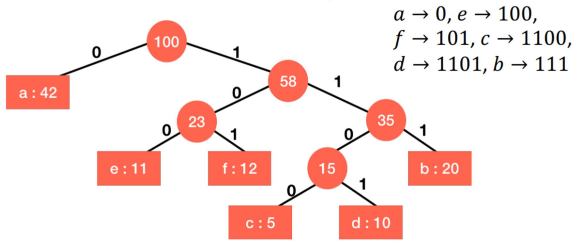
The 26 English letters can be represented with 5 bits (00000,00001,00010, …)

- But we can assign shorter codes to more frequent letters for optimality!
- Issue – if we use symbols $(a, b, c) = (0,1,01)$, then if we observe 01, this could be ab or c
- **Prefix-Free Encoding (PFE):** A mapping c of symbols to bits where $c(x)$ is not a prefix of $c(y)$ for any labels x, y

Given n symbols with frequencies w_i, find a PFE with encoding lengths l_i that minimizes $L = \sum_{i=1}^{n} w_i l_i$

- Note that L is the length of the compressed text

eg. Let $(w_a, w_b, w_c, w_d, w_e, , w_f) = (42{,}20{,}5{,}10{,}11{,}12)$.
The optimal PFE forms a binary tree:



The leaves are the symbols and their weights
The non-leaves are the sum of their children's weights

Huffman Coding: The $\mathcal{O}(n \log n)$ algo to build this tree

1) Initialize a priority queue Q of symbols, where priority is smallest weight
2) While length$(Q) \geq 2$:
   a. Pop 2 symbols with lowest weights, $(i, w_i), (j, w_j)$
   b. Create node with weight $w_i + w_j$ and children i, j
   c. Insert node back into priority queue

Proof of optimality (via induction).
Consider the following lemmas:

- If $w_1 < w_2$, then $l_1 \geq l_2$ in any OS *(ie. if a letter's frequency is less, its encoded length is not less)*
    - Suppose otherwise, that $w_1 < w_2$ and $l_1 < l_2$
    - Then $w_1 l_2 < w_1 l_1$ and $w_2 l_1 < w_2 l_2$
    - Thus L is not minimal (ie. optimal), a contradiction!
- There exists an OS where the 2 symbols of lowest frequency are siblings
    - Take symbol i with minimal w_i from any OS
    - If l_i isn't maximal, there exists a symbol that's tied in weight but longer in length
    - Swap i with the maximal symbol
        - This doesn't change L as both symbols have the same weight.
    - i must have a sibling
        - If there is no sibling, the extra 0/1 at the end of the encoding has no 1/0 counterpart
        - Then the extra 0/1 is redundant and non-optimal, a contradiction!
    - As i is maximal, this sibling is not a non-leaf node *(whose children would have longer encodings than i)*
    - If i's sibling is not the 2nd lowest-weighted symbol, swap the sibling with it!
        - This doesn't change L due to similar reasons as above.

We claim the GS is optimal with k symbols for any $k \geq 2$
<u>Base Claim:</u> For $k = 2$, assigning 0 and 1 to either of the two symbols is trivially optimal.
<u>Inductive Step:</u> Assume the GS is optimal with k symbols, show the GS is optimal with $k+1$ symbols

- Let i, j be the symbols with the smallest w (and thus longest l) that're combined in GS's 1st step
- Let G be the GS tree with $k+1$ symbols
- Let O be an OS tree with $k+1$ symbols where i, j are siblings
- Let G' be G except we merge i, j into a single symbol x with weight $w_x = w_i + w_j$ and length l_x
- Let O' be O except we merge i, j into a single symbol x with weight $w_x = w_i + w_j$ and length l_x
- Since G', O' have k symbols, by the induction hypothesis, $L_{G'} \leq L_{O'}$
- The sum of the wl terms of the two "new" symbols is equal to $(w_i + w_j)(l_x + 1) = w_x l_x + (w_i + w_j)$
- $L_G = L_{G'} + (w_i + w_j)$
- $L_O = L_{G'} + (w_i + w_j)$
- Thus $L_G \leq L_O$

Dynamic Programming: AKA "memoization", breaking problems into simpler subproblems, solving them recursively or iteratively, and storing solutions to be looked up.

- Divide and conquer is a special case where the subproblems don't overlap.

Top-Down Algorithm: An algorithm that solves the largest subproblem with recursion. Good if...

- Not all subproblems need to be solved
- We don't care about order in solving subproblems

Bottom-Up Algorithm: An algorithm that solves smallest subproblems first, then goes up iteratively. Good if…

- All subproblems have to be solved
- We care about speed and memory – less recursive call overheads, sometimes we can free memory earlier

eg. **Weighted interval scheduling** – finding a subset of non-conflicting intervals with the highest weights

Assume jobs i are ordered by finish time
Define $p(i)$ as the largest $i^* < i$ such that $i^*_{\text{finish}} \le i_{\text{start}}$

- "The closest compatible job to i"
- Can be computed via binary search in $\mathcal{O}(\log n)$

Let $\text{OPT}(i) =$ max total weight of compatible intervals in $\{1, \dots, i\}$ (ie. the optimal solution for the jobs up to i)
Base Case: $\text{OPT}(0) = 0$
Recursive Case: There are two cases:

- If i is in the OS, $\text{OPT}(i) = i_{\text{weight}} + \text{OPT}(p(i-1))$
- If i is not in the OS, $\text{OPT}(i) = \text{OPT}(i-1)$

The equation below (a **Bellman Equation**) describes how to optimally choose a decision to maximize a score.

$$\text{OPT}(i) = \begin{cases} 0 & i = 0 \\ \max\{\text{OPT}(i-1), i_{\text{weight}} + \text{OPT}(p(i-1))\} & i > 0 \end{cases}$$

Below is a $\mathcal{O}(n \log n)$ algorithm – sorting and finding p values is $\mathcal{O}(n \log n)$, and the recursion is $\mathcal{O}(n)$ time. This is also called a **top-down algorithm** as it performs recursion starting at the highest values.

```
def opt(jobs):
    sort(jobs) # also renumber jobs
    p = [i: p(i) for i in range(len(jobs))] # p(i) is done with binary search
    m = [0] + [None] * (len(jobs)  - 1)
    m[len(jobs) - 1] = _opt(m, p, jobs, len(jobs) - 1)
    return m

def _opt(m, p, jobs, i):
    if m[i] is None:
        m[i] = max(_opt(m, p, i - 1), job[i].weight + _opt(p[i]))
    return m[i]
```

Here is an alternate **bottom-up algorithm** that performs recursions starting at the lowest values.

```
def opt(jobs):
    sort(jobs) # also renumber jobs
    p = [i: p(i) for i in range(len(jobs))] # p(i) is done with binary search
    m = [0] + [None] * (len(jobs)  - 1)
    for i in range(len(jobs)):
         m[i] = max(m[i - 1], jobs[i].weight + m[p[i]])
    return m
```

To find the actual subset, we use the equation

$$S(i) = \begin{cases} \emptyset & \text{if } j = 0 \\ S(i-1) & \text{else if } \text{OPT}(i-1) < i_{\text{weight}} + \text{OPT}(p(i)) \\ \{i\} \cup S(p(i)) & \text{else} \end{cases}$$

Calculate $p, \text{OPT}(i)$ first, then calculate $S(i)$, which takes $\mathcal{O}(n)$ time eg. **Knapsack problem** – n items with value and weight. Knapsack with weight capacity W – pack it with the subset of highest-valued items, where total weight is $\leq W$

Let $\mathrm{OPT}(i, w)$ = the maximum value of items $\{1, \dots, i\}$ in a bag of capacity w

$$\mathrm{OPT}(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ \mathrm{OPT}(i-1,w) & \text{if } i_{\text{weight}} > w \\ \max\{\mathrm{OPT}(i-1,w), i_{\text{value}} + \mathrm{OPT}(i-1, w - i_{\text{weight}})\} & \text{otherwise} \end{cases}$$

With memorization, the running-time is $\mathcal{O}(nW)$

Let $\mathrm{OPT}(i, v)$ = the minimum volume needed to pack only items $\{1, \dots, i\}$ with value $\geq v$

$$\mathrm{OPT}(i,v) = \begin{cases} 0 & \text{if } v \leq 0 \\ \infty & \text{else if } i = 0 \\ \min\{\mathrm{OPT}(i-1,v), i_{\text{weight}} + \mathrm{OPT}(i-1, v - i_{\text{value}})\} & \text{otherwise} \end{cases}$$

Calculating $\max\{v \colon \mathrm{OPT}(n, v) \leq W\}$ takes $\mathcal{O}(nV)$ time where $V = 1_{\text{value}} + \dots + n_{\text{value}}$

Therefore, the algorithm is $\mathcal{O}(n \min\{V, W\})$ time.

eg. **Single-source shortest paths** – consider graph $G = (V, E)$ with edge lengths, where $|V| = n$; for each node, find the length of the shortest path P to a given source node s.

Assume G has no cycles with negative total length (otherwise, P's length can be $-\infty$!)

Consider a simple *(no cycle)* shortest path from s_1 to s_n, that is $s_1 \to \dots \to s_{n-1} \to s_n$ – this much exist!
Then $s_1 \to \dots \to s_{n-1}$ must be the shortest path from s_1 to s_{n-1}.

- Otherwise, a shorter path exists from s_1 to s_{n-1}, which we would've taken before getting to s_n

We use a recurrence relation! The shortest P from s_1 to s_n can be found with the shortest P from s_1 to s_{n-1}

Let $\mathrm{OPT}(t, i)$ = length of shortest P connecting node s to t with $\leq i$ edges

- If $P_{\text{length}} \leq i - 1$, then $\mathrm{OPT}(t,i) = \mathrm{OPT}(t, i-1)$
- If $P_{\text{length}} = i$, then $\mathrm{OPT}(t,i) = \min_{v \in V}[\mathrm{OPT}(v, i-1) + (v, i-1)_{\text{length}}]$

So the recurrence relation is

$$\mathrm{OPT}(t,i) = \begin{cases} \min\left\{\mathrm{OPT}(t,i-1), \min\limits_{v \in V}[\mathrm{OPT}(v,i-1) + (v,i-1)_{\text{length}}]\right\} & \text{if } i \neq 0 \\ 0 & \text{else if } t = s \\ \infty & \text{else} \end{cases}$$

There are $\mathcal{O}(n^2)$ possible edges and thus values to compute and thus this many recursive calls.

- This is also the space complexity, but it can be reduced to $\mathcal{O}(n)$ with bottom-up approach
- Calculate $\mathrm{OPT}(t, i)$ for i in increasing order, delete $\mathrm{OPT}(t, i)$ columns once not needed

Calculating the recurrence is $\mathcal{O}(n)$ as we find a minimum value from all nodes.
Thus the running time is $\mathcal{O}(n^3)$ – though there are much better solutions.

eg. **All-pairs shortest paths** – same as single-source shortest paths, but for all every possible source node.

Let $\mathrm{OPT}(s, t, i)$ = length of shortest P connecting node s to t with intermediate nodes in $\{1, \dots, i\}$

- Assume $s \to \dots \to 1 \to \dots \to i \to \dots \to t$ is the potential path
- If i is not used, then $\mathrm{OPT}(s,t,i) = \mathrm{OPT}(s,t,i-1)$
- If i is used, then $\mathrm{OPT}(s,t,i) = \mathrm{OPT}(s,i,i-1) + \mathrm{OPT}(i,t,i-1)$

There are $\mathcal{O}(n^3)$ possible values to compute, and thus this many recursive calls
Calculating the recurrence is $\mathcal{O}(1)$.
Thus the running time is $\mathcal{O}(n^3)$.

eg. **Chain matrix product** – find the minimum operations needed for $M_1 \cdots M_n$ where M_i has shape $d_{i-1} \times d_i$

Let $\mathrm{OPT}(i,j)$ = minimum operations needed to compute $M_i \cdots M_j$

$$\mathrm{OPT}(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{k \in \{i,\dots,j-1\}} \mathrm{OPT}(i,k) + \mathrm{OPT}(k+1,j) + d_{i-1}d_k d_j & \text{if } i < j \end{cases}$$

Calculating $d_{i-1}d_k d_j$ is $\mathcal{O}(n)$, while there are $\mathcal{O}(n^2)$ total calls, so $\mathcal{O}(n^3)$ time.

eg. **Edit distance / sequence alignment** – find how similar two strings $X = [X_1, \dots, X_m], Y = [Y_1, \dots, Y_n]$ are in terms of the minimum deletions /replacements to get from the longer string to the shorter string.

Let $d(a)$ be the cost of deleting a
Let $r(a,b)$ be the cost of replacing a with b
Let $\mathrm{OPT}(i,j)$ = edit distance between $X[:i], Y[:j]$

$$\mathrm{OPT}(i,j) = \begin{cases} 0 & \text{if } i = j = 0 \\ d(Y[j]) + \mathrm{OPT}(i, j-1) & \text{if } i = 0, j > 0 \\ d(X[i]) + \mathrm{OPT}(i-1, j) & \text{if } i > 0, j = 0 \\ \min \begin{Bmatrix} d(x_i) + \mathrm{OPT}(i-1,j) \\ d(y_j) + \mathrm{OPT}(i,j-1) \\ r(x_i,y_j) + \mathrm{OPT}(i-1,j-1) \end{Bmatrix} & \text{else} \end{cases}$$

There are $\mathcal{O}(nm)$ entries in total, and constant-time computation, so $\mathcal{O}(nm)$ time complexity.
But space complexity can be reduced to $\mathcal{O}(\min(n,m))$ with a bottom-up approach – by iterating through rows/columns (which is shorter), only keeping a previous row/column

eg. **Travelling salesman** – Find a minimum "cost" Hamiltonian cycle (cycle that visits all nodes exactly once).

Let v_1 be an arbitrary node in the graph.
The idea is to find the smallest-costing parent of v_i every time.
Let $\mathrm{OPT}(S, v_i)$ = minimum cost distance from v_1 to v_i that visits every node in S once.

$$\mathrm{OPT}(S, v_i) = \min_{v \in S \setminus v_i} (\mathrm{OPT}(S \setminus \{v_i\}, v) + (v, v_i)_{\text{cost}})$$

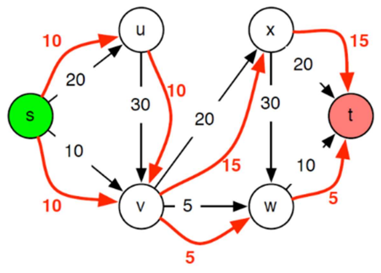
The solution is to calculate

$$\min_{v \in \{v_2, \dots, v_n\}} [\mathrm{OPT}(\{v_2, \dots, v_n\}, v) + (v, v_1)_{\text{cost}}]$$

There are 2^{n-1} possible sets (n nodes minus a source node, each can be in set or not in set) in the table
There are n possible nodes in the table
Each call is $\mathcal{O}(n)$ as we iterate over S, so the running time is thus $\mathcal{O}(n^2 2^n)$

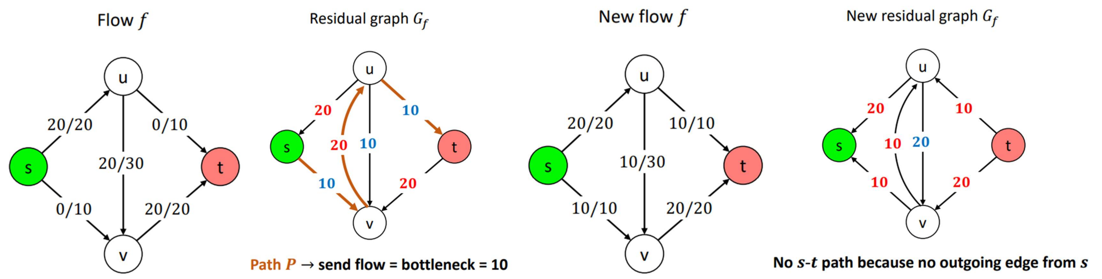
Network Flow: Given a directed graph with edge "capacities" and source/target nodes s, t, find the max "flow" f^*

- Think of it like water moving in pipes of some sizes from s to t – find max possible volume of water flowing
- Despite being a single problem, is able to be generalized to many problems
- Few assumptions:
  - Nothing points to s, and t points to nothing
  - Capacities are non-negative integers (for now)
- ***s-t* Flow:** Of an edge, the "amount of material" carried on it, e_{flow}
  - $0 \leq e_{\text{flow}} \leq e_{\text{capacity}}$ ("capacity constraints")
  - Let $v_{\text{flow in}} = \sum_{e \text{ to } v} e_{\text{flow}}$
  - Let $v_{\text{flow out}} = \sum_{e \text{ from } v} e_{\text{flow}}$
  - $v_{\text{flow in}} = v_{\text{flow out}}$ *(for $v \neq s, t$)* ("flow conservation")
  - We want to maximize $f = s_{\text{flow out}} = t_{\text{flow in}}$
- Greedy approach fails because when it chooses pipes to increase flow to, it can't "undo" a mistake



Bottleneck: The smallest capacity across all edges in a path P from s to t
Residual Graph: Of graph $G = (V, E)$ with flow f, the graph $G_f = (V, E')$

- For each $e = (u, v) \in E$,
  - Add the forward edge $e^{\text{for}} = (u, v) \in E'$ with capacity $e_{\text{capacity}} - e_{\text{flow}}$ if it is > 0 (unsaturated)
  - Add the reverse edge $e^{\text{rev}} = (v, u) \in E'$ with capacity e_{flow} if it is > 0 (used)
- The motivation is being able to send "negative/reverse" flow, partially undoing previous bad moves
- **Augment:** The "sending" of k units of flow through path P by changing $e^{\text{for}}/e^{\text{rev}}$ by $k/-k$ for each $e \in P$.
  - If we increase flow for some $e \in P$,
    - $k \leq e_{\text{capacity}} - e_{\text{flow}}$
    - The new flow is $e_{\text{flow}} + k \leq e_{\text{flow}} + (e_{\text{capacity}} - e_{\text{flow}}) = e_{\text{capacity}}$ *(satisfies constraint)*
  - If we decrease flow for some $e \in P$,
    - $-k \geq -e_{\text{flow}}$
    - The new flow is $e_{\text{flow}} - k \geq e_{\text{flow}} - e_{\text{flow}} = 0$ *(satisfies constraint)*
  - $e_{\text{flow in}} = e_{\text{flow out}}$ for all $e \in P$
    - Each node $v \in P \setminus \{s, t\}$ has two edges touching it
    - If $\ldots \to v \to \cdots$, $e_{\text{flow in}}$ and $e_{\text{flow out}}$ both increase/decrease by k.
    - If $\ldots \leftarrow v \to \cdots$, $e_{\text{flow out}}$ has $+k$ (*forward edge*), then $-k$ (*reverse edge*), so the same flow.
    - If $\ldots \to v \leftarrow \cdots$, $e_{\text{flow in}}$ has $+k$ (*forward edge*), then $-k$ (*reverse edge*), so the same flow.



Ford-Fulkerson (FF) Algorithm: Max flow algorithm, in $\mathcal{O}\big((n+m)C\big)$ where $C = \sum_{e \text{ from } s} e_{\text{capacity}}$

```
def max_flow(G, s, t):
    for e in G.E:
        e.flow = 0

    P = find_path(residual_graph(G), s, t) # find_path is DFS
    while len(P) > 0:
        k = bottleneck(P)
        augment(P, k)
        P = find_path(residual_graph(G), s, t)
    return s.flow
```

ResidualGraph returns a graph. $\mathcal{O}(n+m)$

- Once created, no need to recreate G_f later – only update edges! So it's $\mathcal{O}(m)$ in the loop.

FindPath returns a list of edges. DFS is $\mathcal{O}(n+m)$

Bottleneck returns an integer. $\mathcal{O}(m)$

Augment mutates every edge's flow. $\mathcal{O}(m)$

Bottleneck is ≥ 1, so flow increases per loop
Max possible flow is $C = \sum_{e \text{ from } s} e_{\text{capacity}}$
Loop body is $\mathcal{O}(n+m)$, so $\mathcal{O}\big((n+m)C\big)$

- If edge capacities are integers, FF's solution has integer edge flows *(Integrality Theorem)*
- Issue is C can be exponential!
  - If we choose the path with the maximum bottleneck, $\mathcal{O}(m^2 \log C)$
  - If we choose the shortest path with BFS, $\mathcal{O}(nm^2)$
- ***s-t* Cut:** A partition (A, B) of V (i.e. $A \cup B = V, A \cap B = \emptyset$)
- **Capacity:** Of a cut, the value $\text{cap}(A, B) = \sum_{e \text{ from } A} e_{\text{capacity}}$
  - $f \leq \text{cap}(A, B)$
- **Max Flow-Min Cut Theorem:** $\min_{(A,B)} \text{cap}(A, B) = \max f = f^*$
  - The min-cut is the cut that minimizes LHS
  - Min-cut can be found by finding f^* with FF, constructing G_{f^*}, finding the set of nodes reachable from s (use BFS)
  - To find visually, draw a curve intersecting saturated edges

$$\begin{aligned} f &= A_{\text{flow out}} - A_{\text{flow in}} \\ &\leq A_{\text{flow out}} \\ &= \sum_{e \text{ from } A} e_{\text{flow}} \\ &\leq \sum_{e \text{ from } A} e_{\text{capacity}} \\ &\leq \text{cap}(A, B) \end{aligned}$$

eg. Prove the FF algorithm is correct.

For any s-t cut (A, B) and flow, we have $f \leq \text{cap}(A, B)$
Therefore $\max f \leq \min_{(A,B)} \text{cap}(A, B)$

Let $f^* =$ flow returned by FF.
We will show $f^* = \text{cap}(A, B)$ for some s-t cut (A, B).
Let $A^* =$ nodes reachable from s in G_f
Let $B^* = V \setminus A^*$
Then (A^*, B^*) form a s-t cut of G_f

Let $S_1 = \{\text{edges from } A^*\}$
Edges in S_1 satisfy $e_{\text{flow}} = e_{\text{capacity}}$
- Otherwise, $e_{\text{capacity}} - e_{\text{flow}} > 0$
- So FF adds $e^{\text{for}} = (u, v) \in E^*$
- As $u \in A^*$ and v is reachable from u, then $v \in A^*$
- Then $e \notin S_1$, a contradiction

Let $S_2 = \{\text{edges to } A^*\}$
Edges in S_2 satisfy $e_{\text{flow}} = 0$
- Otherwise, $e_{\text{flow}} > 0$
- So FF adds $e^{\text{rev}} = (v, u) \in E^*$
- As $u \in A^*$ and v is reachable from u, then $v \in A^*$
- Then $e \notin S_2$, a contradiction

So we have

$$\begin{aligned}\sum_{e \in S_1} e_{\text{flow}} &= \sum_{e \in S_1} e_{\text{capacity}} \\ &= \sum_{e \text{ from } A^*} e_{\text{capacity}} \\ &= \text{cap}(A^*, B^*)\end{aligned}$$

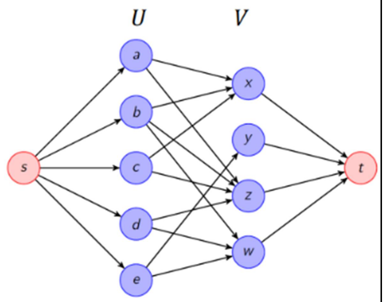
$$\begin{aligned}\sum_{e \text{ to } A^*} e_{\text{flow}} &= \sum_{e \in S_2} e_{\text{flow}} \\ &= \sum_{e \in S_2} 0 \\ &= 0\end{aligned}$$

$$\begin{aligned}\therefore f^* &= s_{\text{out}} \\ &= \sum_{e \text{ from } A^*} e_{\text{flow}} - \sum_{e \text{ to } A^*} e_{\text{flow}} \\ &= \text{cap}(A^*, B^*) - 0 \\ &= \text{cap}(A^*, B^*)\end{aligned}$$

Therefore, $f^* = \text{cap}(A^*, B^*) \geq \min_{(A,B)} \text{cap}(A, B) \geq \max f$, so FF is optimal!

You can prove $\min_{(A,B)} \text{cap}(A, B) = \max f$ using the FF solution, but I don't care enough.

eg. **Bipartite matching** – find a maximally-sized matching, a set of edges without common vertices, in a bipartite graph $G = (U \cup V, E)$.



Construct a directed flow graph from G with weight 1 edges:
- Point s to everything in U
- Align all edges in U to V to point to V
- Point everything in V to t

Notice that $f^* = k \Leftrightarrow G$ is a matching of size k
- This is since $s \xrightarrow{1} u_{1,\ldots,k} \xrightarrow{1} v_{1,\ldots,k} \xrightarrow{1} t$
- Capacity of 1 prevents flow from "splitting" into two or "combining" into one (ie. common vertices)

Since $C = \sum_{e \text{ leaving } s} 1 = |U| = \mathcal{O}(n)$ and edge count is $m \in \mathcal{O}(n^2)$, then running-time on FF is $\mathcal{O}(n^3)$
If all edge capacities, Dinitz' algorithm is an algorithm that runs in $\mathcal{O}(m\sqrt{n})$ time.

eg. **Multiple sources/sinks** – Find maximum flow from sources to sinks

Let the sources/sinks be $s_1, \ldots, s_n$ and $t_1, \ldots, t_m$
Add a source s/sink t that points to all sources/sinks with capacity ∞
Note that $C = \sum_{i=1}^{n} \sum_{e \text{ from } s_i} e_{\text{capacity}}$, so the running time is not ∞.

eg. **Edge-disjoint paths** – find max number of $s \to t$ paths that don't share edges

Done by setting $e_{\text{capacity}} = 1$ for all edges. $\exists G$ with $f^* = k \Leftrightarrow$ there are k edge-disjoint paths $P_1, \dots, P_k$

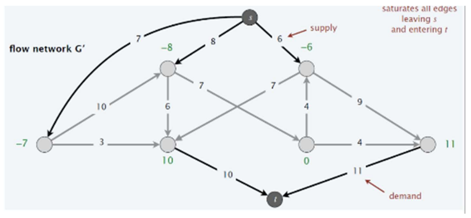
- Show $f^* = k \Rightarrow$ there are k edge-disjoint paths
    - Since $f^* = k$, then k edges from s has flow 1
    - Consider any such edge (s, v_1). v_1 must flow to 1 other vertex, which eventually flows to t
    - This must hold for all k edges – the edges picked are unique each time due to capacity 1
    - Thus, the path is edge-disjoint
- Show there are k edge-disjoint paths $\Rightarrow f^* = k$
    - Since the k paths are edge-disjoint, there exists $\geq k$ edges pointing from s
    - Consider a network with $e_{\text{flow}} = \mathbb{I}(e \in P_i)$
    - Since paths are edge-disjoint, flow conservation and capacity constraints hold
    - So the network has $s_{\text{flow out}} = t_{\text{flow in}} = k$, meaning $f^* = k$

FF either finds a network with $f^* = k$, which will have k edge-disjoint paths; or finds nothing, in which case there is not k edge-disjoint paths.

eg. **Circulation** – Nodes have "demands", the amount of flow to take out of v (if $v_{\text{demand}} > 0$, a "demand/surplus node") or put into v (if $v_{\text{demand}} < 0$, a "supply/deficit node"). We need $\sum_{e \text{ to } v} e_{\text{flow}} - \sum_{e \text{ from } v} e_{\text{flow}} = v_{\text{demand}}$

Create a source s and sink t

- For each deficit node v, add (s, v) with capacity $|v_{\text{demand}}|$
- For each surplus node v, add (v, t) with capacity $|v_{\text{demand}}|$
- So $s_{\text{flow out}} \leq \sum_{e \text{ from } s} e_{\text{capacity}} = \sum_{v \in V : v_{\text{demand}} < 0} |v_{\text{demand}}|$
- So $t_{\text{flow in}} \leq \sum_{e \text{ to } t} e_{\text{capacity}} = \sum_{v \in V : v_{\text{demand}} > 0} v_{\text{demand}}$

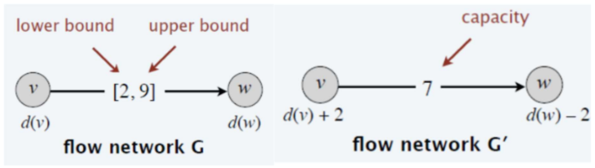


The network has a circulation $\Leftrightarrow f^* = \sum_{v \in V : v_{\text{demand}} > 0} v_{\text{demand}} = \sum_{v \in V : v_{\text{demand}} < 0} |v_{\text{demand}}|$

eg. **Circulation with lower bounds** – Flow has a lower bound

For any edge $e = (u, v)$ with flow $[e_{\text{min flow}}, e_{\text{capacity}}]$,

- Set $e_{\text{capacity}} \leftarrow e_{\text{capacity}} - e_{\text{min flow}}$
- Set $u_{\text{demand}} \leftarrow u_{\text{demand}} + e_{\text{min flow}}$
- Set $v_{\text{demand}} \leftarrow v_{\text{demand}} - e_{\text{min flow}}$



u_{demand} increases to force at least $e_{\text{min flow}}$ units of flow past u to v
v_{demand} decreases to "offset" this.
A flow of $e_{\text{flow}} - e_{\text{min flow}}$ out of $e_{\text{capacity}} - e_{\text{min flow}}$ in augmented graph is equivalent to
A flow of e_{flow} out of e_{capacity} in original graph

eg. **Survey design** – Survey with m products, n consumers, one question per product. Need to ask product i's question to $[p_i, p_i']$ customers, ask customer i between $[c_i, c_i']$ questions only about products they own, O_i.

If $c_i = c_i' = p_i = p_i' = 1$, we can use bipartite matching
Create a source and sink:

- For every consumer i, create edge (s, i) with flow $[c_i, c_i']$
- For every consumer i to product j, create edge (i, j) with flow [0,1]
- For every product j, create edge (j, t) with flow $[p_j, p_j']$
- Create edge (t, s) with flow $[0, \infty]$
- Set surpluses/deficits to 0

There is a feasible survey $\Leftrightarrow$ there is a circulation in the network eg. **Image segmentation** – Separate foreground and background, given an image (2D array of pixels), likelihood estimates a_i/b_i of pixel i being foreground/background, and penalty $p_{i,j}$ for not classifying neighbours i, j the same

Let E be the edge set of all neighbouring pairs of pixels
Let A, B be set of pixels labelled foreground/background
Divide pixels V into a *s-t* cut (A, B) that minimize the loss function

$$\sum_{i\in A} b_i + \sum_{i\in B} a_i + \sum_{(i,j)\in E: i\in A\setminus\{s\}, j\in B\setminus\{t\}} p_{i,j}$$

Create a source and sink,

- For each pixel, create a node v_i plus edges $(s, v_i), (v_i, t)$ with capacities a_i, b_i
- For all neighbouring pixels (i, j), create edges $(v_i, v_j), (v_j, v_i)$ with capacities $p_{i,j}$

If we include $s \in A$, then

$$\begin{aligned}\text{cap}(A, B) &= \sum_{e \text{ from } A} e_{\text{capacity}} \\ &= \sum_{e=(i,j): i\in A, j\in B} \begin{cases} a_j & i = s \\ b_i & j = t \\ p_{i,j} & \text{else} \end{cases} \\ &= \sum_{(s,j): j\in B} a_j + \sum_{(i,t): i\in A} b_i + \sum_{(i,j): i\in A\setminus\{s\}, j\in B\setminus\{t\}} p_{i,j} \\ &= \sum_{i\in B} a_i + \sum_{i\in A} b_i + \sum_{(i,j): i\in A\setminus\{s\}, j\in B\setminus\{t\}} p_{i,j}\end{aligned}$$

Thus, run FF to find the min-cut, which minimizes $\text{cap}(A, B)$ which is equal to the loss function.

eg. **Profit maximization** – n tasks each with profit $p_i \in \mathbb{R}$ and precedence relations E, where $(i, j) \in E$ means if i is done, j must be done. Find a subset of tasks that maximizes profit

Represent input as a directed graph

- Vertices are tasks
- Vertex weights are profits
- Edges are precedence constraints

Add a sink and source

- For each positively-weighted vertex v_i, add edge (s, v_i) with capacity p_i
- For each negatively-weighted vertex v_i, add edge (v_i, t) with capacity $|p_i|$
- Set pre-existing edges' capacities to ∞

Then

$$\begin{aligned}\text{cap}(A, B) &= \sum_{e \text{ from } A} e_{\text{capacity}} \\ &= \sum_{e=(i,j): i\in A, j\in B} \begin{cases} |p_i| & j = t \\ p_j & i = s \\ \infty & \text{else} \end{cases} \\ &= \sum_{(s,j): j\in B} p_j + \sum_{(i,t): i\in A} |p_i| + \sum_{(i,j): i\in A\setminus\{s\}, j\in B\setminus\{t\}} \infty\end{aligned}$$

A finite-capacity cut exists because $A = \{s\}$ is always an option – we can pretend the last term won't exist.
The last summation must be 0, which enforces precedence constraints

$$\begin{aligned}\text{profit} = \sum_{i\in A} p_i &= \sum_{(s,i): i\in A} p_i - \sum_{(i,t): i\in A} |p_i| \\ &= \left(\sum_{(s,i): i\in A} p_i + \sum_{(s,j): j\in B} p_j\right) - \left(\sum_{(i,t): i\in A} |p_i| + \sum_{(s,j): j\in B} p_j\right) \\ &= \sum_{(s,i): i\in V} p_i - \text{cap}(A, B)\end{aligned}$$

Since the 1st term is a constant, minimizing $\text{cap}(A, B)$ maximizes profit.

Linear Programming: Maximize linear functions $f\colon \mathbb{R}^n \to \mathbb{R}$ with $f(\mathbf{x}) = \mathbf{a}^{\mathrm{T}}\mathbf{x}$ given m linear constraints, $x_i \geq 0$

- Constraints are of form $\mathbf{a}^{\mathrm{T}}\mathbf{x} \leq c$, but can be applied to other forms:

If $\mathbf{a}^{\mathrm{T}}\mathbf{x} = c$	$\mathbf{a}^{\mathrm{T}}\mathbf{x} \leq c, \mathbf{a}^{\mathrm{T}}\mathbf{x} \geq c$
If $\mathbf{a}^{\mathrm{T}}\mathbf{x} \geq c$	$-\mathbf{a}^{\mathrm{T}}\mathbf{x} \leq -c$
If $\mathbf{a}^{\mathrm{T}}\mathbf{x}$ is minimized	$-\mathbf{a}^{\mathrm{T}}\mathbf{x}$ is maximized
If x_i are unbounded	Set $x_i = x_i' - x_i''$, where $x_i' \geq 0$ and $x_i'' \geq 0$
If $\lvert x_i \rvert \leq c$	$x_i \leq c, -x_i \leq c$
If $\lvert x_i \rvert \geq c$	Solve for case $x_i \geq c$, case $x_i \leq -c$
If $f(\lvert x_i \rvert)$ is minimized	$f(t)$ is minimized, where $t \geq x_i$ and $t \geq -x_i$ (ie. $t \geq \lvert x_i \rvert$)
If $f(\lvert x_i \rvert)$ is maximized	$f(t)$ is maximized, where $t \leq x_i$ and $t \leq -x_i$ (ie. $t \leq \lvert x_i \rvert$)

- **Half-Space:** A region bisected by a linear function, represented by $\mathbf{a}^{\mathrm{T}}\mathbf{x} \leq c$
- **Objective Function:** The function to minimize/maximize
- **Feasible Region/Polytope:** The region of all possible solution points – an intersection of half-spaces
- **Convex Set:** A set if $\forall x, y \in S, \alpha \in [0,1], \alpha x + (1-\alpha)y \in S$ – *"for any points, the line between them is in S"*
    - **Vertex:** "Protrusion points" in S – can't be written as $\alpha x + (1-\alpha)y$
- Feasible regions are convex sets
- The solution is a vertex
- Solution may not exist if:
    - Constraints are infeasible – the feasible region is empty
    - The problem is unbounded – the variable can be infinitely large/small

The **primal** is a maximization problem,

$$\begin{aligned} \max x_1 + 6x_2 \\ x_1 \leq 200 \\ x_2 \leq 300 \\ x_1 + x_2 \leq 400 \\ x_1, x_2 \geq 0 \end{aligned}$$

Add variables $y_i \geq 0$ for each of the "main" constraints, like

$$\begin{aligned} x_1 y_1 \leq 200 y_1 \\ x_2 y_2 \leq 300 y_2 \\ (x_1 + x_2) y_3 \leq 400 y_3 \end{aligned}$$

Add the equations up to produce

$$(y_1 + y_3)x_1 + (y_2 + y_3)x_2 \leq 200y_1 + 300y_2 + 400y_3$$

We want

$$x_1 + 6x_2 \leq (y_1 + y_3)x_1 + (y_2 + y_3)x_2$$

So we need $y_1 + y_3 \geq 1$ and $y_2 + y_3 \geq 6$
This gives us the **dual**, an equivalent minimization problem,

$$\begin{aligned} \min 200y_1 + 300y_2 + 400y_3 \\ y_1 + y_3 \geq 1 \\ y_2 + y_3 \geq 6 \\ y_1, y_2, y_3 \\ \geq 0 \end{aligned}$$

The standard form of an LP, the primal, is

$$\begin{aligned} \max \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ \mathbf{x} \geq 0 \end{aligned}$$

The form of a dual is

$$\begin{aligned} \min \mathbf{y}^{\mathrm{T}}\mathbf{b} \\ \mathbf{y}^{\mathrm{T}}\mathbf{A} \geq \mathbf{c}^{\mathrm{T}} \\ \mathbf{y} \geq 0 \end{aligned}$$

Weak Duality Theorem: For feasible $\mathbf{x}, \mathbf{y}$, we have $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq (\mathbf{y}^{\mathrm{T}}\mathbf{A})\boldsymbol{x} = \mathbf{y}^{\mathrm{T}}(\mathbf{A}\boldsymbol{x}) = \mathbf{y}^{\mathrm{T}}\mathbf{b}$
Strong Duality Theorem: For optimal $\mathbf{x}, \mathbf{y}$, we have $\mathbf{c}^{\mathrm{T}}\mathbf{x} = \mathbf{y}^{\mathrm{T}}\mathbf{b}$

- Optimality can be proved by finding an upper/lower bound by doing algebra on the constraints
- Or find a dual solution equal to a given primal solution

Simplex Algorithm: Algorithm that finds a vertex, repeatedly moves to neighbours with better objective values

- Simple to state, but difficult to implement
- Number of vertices (ie. running-time) can be exponential
- Can be very fast on some problem classes

eg. Solve a network flow with LP

Capacity constraint is $0 \leq e_{\text{flow}} \leq e_{\text{capacity}}$ for all $e \in E$
Flow conservation is $\sum_{e \text{ from } v} e_{\text{flow}} = \sum_{e \text{ to } v} e_{\text{flow}}$ for $v \in V \setminus \{s, t\}$
Maximize $f = \sum_{e \text{ from } s} e_{\text{flow}}$

eg. **Single-source shortest paths** with LP

For each vertex v, define $v_{\text{dist}} = \begin{cases} \min_{u \in v.\text{childre}} \{u_{\text{dist}} + (u,v)_{\text{length}}\} & \text{else} \\ 0 & v = s \end{cases}$
Constrain $v_{\text{dist}} \le u_{\text{dist}} + (u,v)_{\text{length}}$ for $u \in v.$ children
Constraint $s_{\text{dist}} = 0$
Maximize t_{dist}

eg. **Multicommodity flow –** given graph with edge capacities and k commodities with i_{demand} units that need to be sent from i_{source} to i_{sink}, send k flows each with value i_{demand}

Capacity constraint is $0 \le e_{\text{flow}} \le e_{\text{capacity}}$ for all $e \in E$
Flow conservation is $\sum_{e \text{ from } v} e_{\text{flow}} = \sum_{e \text{ to } v} e_{\text{flow}}$ for $v \in V \setminus \{s,t\}$
Constraint $\sum_{e \text{ from } i_{\text{source}}} e_{\text{flow}} - \sum_{e \text{ from } i_{\text{sink}}} e_{\text{flow}} = i_{\text{demand}}$ for $i \in \{1, \dots, k\}$
Maximize $f = \sum_{e \text{ from } s} e_{\text{flow}}$

Turing Machine TM: A machine consisting of

- A *tape* of finite symbols (ie. the inputs), where intermediate computations and outputs are also written
- A *head pointer* pointing to the input's start
- An *internal state* and *transition function* for how to move states (read/write to tape, move head pointer)
  - Turing machines can only move forward/backward, but this only adds a factor of n to running-time, which doesn't affect polynomial time

Decision Problem: A problem with binary output ("YES" or "NO")
Efficient: An algorithm, if it works in polynomial time, $\mathcal{O}(n^k)$ for some $k \in \mathbb{R}$
Church-Turing Thesis: "Everything computable is computable by TMs"
Extended Church-Turing Thesis: "Everything efficiently computable is efficiently computable by TMs"

P (Polynomial Time): Set of decision problems efficiently computable by a TM

- eg. Addition, multiplication, square root, network flows, shortest paths, FFT, prime checking
- Constants like C in the FF-algorithm are exponential?, but $\log C$ is considered polynomial?

NP (Nondeterministic Polynomial Time): Set of decisions problems that a TM can efficiently confirm "YES" of, given an "example" of correctness. So P is naturally a subset of NP.

- eg. Questions like "Does there exist…", "Is it always false…"
- eg. **Subset sums** – if there exists a zero-sum subset of a given set. Enumerating all subsets is $\mathcal{O}(2^n)$, but verifying that a subset has sum zero is $\mathcal{O}(n)$

co-NP: Same as NP but instead of "YES", it efficiently confirms "NO" given an "example" of wrongness

- eg. Questions like "Does there not exist…", "Is it always true…"

p-Reducible: Decision problem A to decision problem B, notated $A \le_p B$, if there exists a polynomial-time TM that convert any instance of A into B with the same answer

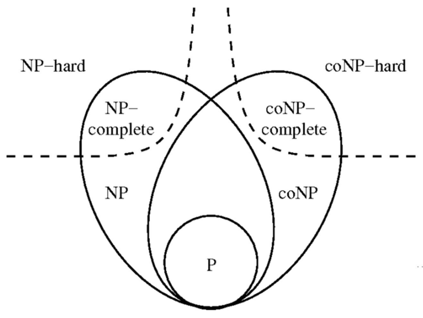
- B is efficiently solvable $\Rightarrow A$ is efficiently solvable
- B is "at least as hard/complex" as A

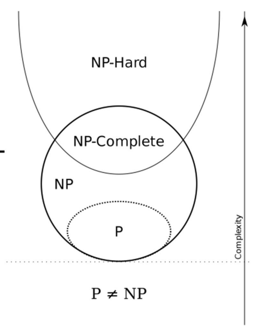
NP-Hard: Decision problem B if $A \le_p B$ for all $A \in$ NP *("at least as complete as everything NP")*
NP-Complete: Decision problem $B \in$ NP if $A \le_p B$ for all $A \in$ NP *("NP, at least as complex as everything NP")*

- B is efficiently solvable $\Rightarrow$ everything in NP is efficiently solvable
  - NP-complete problems are the "hardest problems" in NP
- $B \le_p A$ for some $A \in$ NP $\Rightarrow A$ is also NP-complete
- A is NP-complete $\Rightarrow A$ is NP-hard

Cook's Conjecture: $P \neq NP$

- Widely believed true, since there are many NP-complete problems – if we can solve any of them, we can efficiently solve all problems in NP. But no one has found a polynomial-time solution for them.





Conjunctive Normal Form (CNF): A logical statement of form $\phi = C_1 \wedge \cdots \wedge C_m$, where C_i are clauses

- **Clauses:** Statements of form $C = \ell_1 \vee \cdots \vee \ell_r$, where a literal ℓ is a boolean x or its negation $\overline{x}$
- ***k*-CNF:** CNF where clauses have $\leq k$ literals
- **Exact *k*-CNF:** CNF where clauses have k literals

Satisfiable: A CNF formula ϕ, if some combination of TRUE/FALSE boolean values makes it true

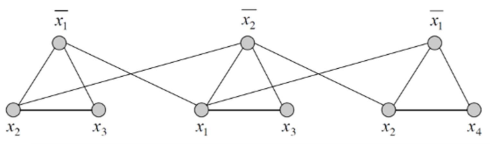
- **SAT**: "Given a CNF formula ϕ, is it satisfiable?"
- **Exact *3*SAT**: "Given an exact 3CNF formula ϕ, is it satisfiable?"
- **Cook-Levin Theorem:** SAT and Exact 3SAT are NP-complete
- The "original" NP-complete problem that reduces to other problems

eg. **Independent set** – does an undirected graph $G = (V, E)$ have a set S of k disconnected vertices?

Independent set is in NP

- Given the actual k-length S, independent-ness can be verified in $\mathcal{O}(n + m)$ time

Exact 3SAT $\leq_p$ Independent set



$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

- Given ϕ with k clauses, construct graph (G, k):
  - For each literal, create a vertex
  - For each clause, connect the 3 vertices in a triangle
  - For each literal, connect it to its negation(s)
- Exact 3SAT = YES $\Rightarrow$ Independent set = YES
  - For each of k clauses, at least one literal ℓ is TRUE – pick any one of them to put in S
  - One literal per clause is chosen $\Rightarrow$ k literals are disconnected
    - x and $\overline{x}$ cannot both be true – the false one won't ever be picked
    - Vertices in different clauses are disconnected
- Independent set = YES $\Rightarrow$ Exact 3SAT = YES
  - k literals are disconnected $\Rightarrow$ one literal per clause (and not both x and $\overline{x}$) is chosen
    - If we choose x and $\overline{x}$, they are connected, a contradiction
    - If we choose ≥ 1 literal per clause, they are connected, a contradiction
    - If we choose 0 literals per clause, we have $< k$ clauses, a contradiction
  - Set everything in S as true and their negations as false
  - We don't care about the other literals – set them to whatever.

Since…

- Exact 3SAT is NP-complete
- Exact 3SAT$\leq_p$ Independent set
- Independent set is in NP

Then independent set is NP-complete.

eg. **Subset sum** – is there a subset $S' \subseteq S = \{w_1, \dots, w_n\}$ with sum W where all are integers?

Subset sum is in NP

- Given the actual subset, can verify the sum in $\mathcal{O}(n)$ time

Exact 3SAT $\leq_p$ Subset sum

- Given ϕ with n clauses, construct a $2(n+3) \times (n+3)$ matrix T:
  - For each ℓ_i and clause C_i, add a column and two rows
  - The column for ℓ_i corresponds to two rows for x_i and $\overline{x_i}$
    - The cells at column ℓ_i are 0, except for at rows x_i and $\overline{x_i}$, where it is 1
  - The column for C_i corresponds to two rows α_i, β_i
    - The cells at column C are 0, except for:
      - At row α_i (= 1)
      - At row β_i (= 2)
      - At row x_i or $\overline{x_i}$, if that boolean being true makes C true (= 1)
- Set $W = 111\underbrace{4 \cdots 4}_{n}$, and w_i = row i read like a number

Consider the 3CNF equation $\phi = C_1 \wedge C_2 \wedge C_3$, where

$$C_1 = \overline{x_1} \vee x_2 \vee x_3$$
$$C_2 = x_1 \vee \overline{x_2} \vee x_3$$
$$C_3 = \overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$$

The constructed table is shown on the right.

	ℓ_1	ℓ_2	ℓ_3	C_1	C_2	C_3	w_i
x_1	1	0	0	0	1	0	100010
$\overline{x_1}$	1	0	0	1	0	1	100101
x_2	0	1	0	1	0	0	10100
$\overline{x_2}$	0	1	0	0	1	1	10011
x_3	0	0	1	1	1	0	1110
$\overline{x_3}$	0	0	1	0	0	1	1001
α_1	0	0	0	1	0	0	100
β_1	0	0	0	2	0	0	200
α_2	0	0	0	0	1	0	10
β_2	0	0	0	0	2	0	20
α_3	0	0	0	0	0	1	1
β_3	0	0	0	0	0	2	2
W	1	1	1	4	4	4	

- For $i \leq 3$,
  - The i-th digit of W is 1
  - To get a sum with the i-th digit equal to 1, choose the w for x_i or $\overline{x_i}$, but not both
- For $i \geq 4$,
  - The i-th digit of W is 4
  - The α, β rows have filler values (add up to 3)
  - This forces previous w values to contribute ≥ 1 to the i-th digit, meaning C_{i-3} will be true!

Continuing with the proof…
- Exact 3SAT = YES ⇒ Subset sum = YES
  - Choose the w corresponding to the values of ℓ_1, ℓ_2, ℓ_3 that are TRUE
  - Choose the right α, β rows so the value of digits $i \geq 4$ are all 4 – then we have found a valid S'!
- Subset sum = YES ⇒ Exact 3SAT = YES
  - Set $x_i, \overline{x_i}$ equal to whether the corresponding w values were picked
  - The $i \leq 3$ constraints ensure x_i and $\overline{x_i}$ are not both true
  - The $i \geq 4$ constraints ensure all C_i are true

Therefore, subset sum is NP-complete.

eg. Simple **Binary Integer Linear Programming (BILP)** – find $x_i \in \{0,1\}$ where $A\mathbf{x} \leq \mathbf{b}$ and A is $m \times n$.

Simple BILP is in NP
- Involves checking all possible 0-1 combinations of the x_i in worst-case, which is $\mathcal{O}(2^n)$
- Given the actual length-n vector $\mathbf{x}$, the inequality can be verified in $\mathcal{O}(nm)$ time

Exact 3SAT $\leq_p$ Independent set
- Given ϕ with n clauses, construct an equation for each clause:
  - Represent x_i as itself
  - Represent $\overline{x_i}$ as $1 - x_i$
  - Represent OR as sum ≥ 1 (eg. $x_i \vee \overline{x_2}$ becomes $x_1 + (1 - x_2) \geq 1$)
- It is easy to verify this 1/0 logic is equivalent to TRUE/FALSE

eg. **Vertex cover** – given undirected graph $G = (V, E)$ is there a k-sized subset $S \subseteq V$ that touches every edge?

S is a vertex cover ⇔ $V \setminus S$ is an independent set
- S is a vertex cover ⇒ $V \setminus S$ is an independent set
  - For all $e = (u, v) \in E$, at least one of $u, v \in S$
  - Thus, at most one of $u, v \in V \setminus S$
  - Thus, $V \setminus S$ has no edges (u, v) where $u, v \in V \setminus S$, so it is independent
- Proof in the opposite direction works the same way!

Vertex cover is in NP

- Involves checking all possible subsets of V in worst-case, which is $\mathcal{O}(2^n)$
- Given the actual k-length S, vertex cover-ness can be verified in $\mathcal{O}(n+m)$ time

Independent set $S \leq_p$ Vertex cover

- Take an independent set S of size $n-k$, and find $V \setminus S$, which is of size k
- Finding one is equivalent to finding the other

eg. **Set cover** – Given a set U and a size-n set S of some subsets of U, is there a size-k subset of S with union U?

Vertex cover is in NP

- Involves checking $\mathcal{O}\left(\binom{n}{k}\right)$ possible combinations of sets from S in the worst-case
- Given the actual k-length subset, union can be verified in $\mathcal{O}(n)$ time

Vertex cover $S \leq_p$ Set cover

- Given a vertex cover S' with graph $G=(V,E)$ of size k, create set cover where:
  - $U = E$
  - For $v \in V$, add $\{e \in E\colon e \text{ touches } v\}$ to S
- Choosing k sets with union U = choosing k vertices S' whose incident edges cover all edges

eg. Prove that SAT $\leq_p$ Exact 3SAT

Given ϕ with n clauses, for each clause, create $z_1, \dots, z_{k-3}$ and the clause:

SAT	*Exact 3SAT*
$\ell_1 \vee \ell_2$	$(\ell_1 \vee \ell_2 \vee z_1)$
$\vee\, \ell_3$	$\wedge\, (\ell_3 \vee \overline{z_1} \vee z_2)$
$\vee\, \ell_4$	$\wedge\, (\ell_4 \vee \overline{z_2} \vee z_3)$
$\vee \cdots$	$\wedge \cdots$
$\vee\, \ell_{k-2}$	$\wedge\, (\ell_{k-2} \vee \overline{z_{k-4}} \vee z_{k-3})$
$\vee\, \ell_{k-1} \vee \ell_k$	$\wedge\, (\ell_{k-1} \vee \ell_k \vee \overline{z_{k-3}})$

If $z_{i-2} = z_{i-1}$, then no constraint on ℓ_i
If $z_{i-2} \neq z_{i-1}$, then ℓ_i must be true

Regardless of z_i, ≥ 1 of the brackets has a "ℓ_i must be true" constraint.
For instance, if TRUE $= z_{\text{even}} \neq z_{\text{odd}}$, then $\ell_{k-1} \vee \ell_k$ must hold.
This ensures the same OR property from SAT holds.

If any ℓ_i hold, some arrangement of z variables makes it true.
If no ℓ_i hold, no arrangement of z variables makes it true.

Approximation: Efficient algorithms that find approximate solutions to (usually optimization) problems.

- $\text{ALG}(I)$ = approximate algorithm's solution, where I = a problem instance
- $\text{OPT}(I)$ = optimal solution
- Maximization/minimization = profit/cost
- **Approximation Ratio:** The value $\frac{\text{OPT}(I)}{\text{ALG}(I)}$ or $\frac{\text{ALG}(I)}{\text{OPT}(I)}$ for maximization/minimization (should be ≥ 1)
- **Worst Case *c*-Approximation Ratio:** If $\text{cost}\big(\text{ALG}(I)\big) \leq c\,\text{cost}\big(\text{OPT}(I)\big)$
- Can involve greedy-fying an algorithm, rounding a real-valued LP, local adjustments to improve solutions

PTAS (Polynomial Time Approximation Scheme): Takes inputs and $\epsilon > 0$, returns $(1+\epsilon)$-approximation

- Running-time is polynomial on n, but not necessarily on $\frac{1}{\epsilon}$

FPTAS (Fully PTAS): Takes inputs and $\epsilon > 0$, returns $(1+\epsilon)$-approximation with running time $\text{poly}(n, \frac{1}{\epsilon})$

eg. **Makespan** – n jobs with processing time t_i, m machines with load $L_i = \sum_{j \in S[i]} t_j$ where $S[i] =$ jobs assigned to machine i. Minimize $\max_{i \in \{1,\dots,m\}} L_i$.

Note even $m = 2$ is NP-hard, use a $\mathcal{O}(n \log m)$ greedy approach:

- Assign each job (n) to a machine with the smallest load so far ($\log m$), using a priority queue

Let L^* = optimal makespan. Note that

- $L^* \geq \max_{j \in \{1,\dots,n\}} t_j$ some machine has to do the longest job
- $L^* \geq \frac{1}{m} \sum_{j=1}^{m} t_j$ at least machine does $\geq$ than the average load

We claim $\text{ALG}(I)$ is a 2-approximation

- Let machine i have the max load, so $L = L_i$
- Let job j^* be the last one scheduled to i; right before scheduling, $L_i - t_{j^*}$ has the minimal load
  - Otherwise, adding t_{j^*} to another machine would have made the load larger than L_i
- Thus $L_i - t_{j^*}$ is smaller than the average load, $\frac{1}{m} \sum_{j=1}^{m} t_j \leq L^*$
- Therefore, $L = L_i \leq L^* + t_{j^*} \leq 2L^*$
- Technically, make a stronger claim that $L = 2L^* - \frac{1}{m}$
  - We have $L_i \leq t_{j^*} + \frac{1}{m-1} \sum_{j \in \{1,\dots,m\} \setminus \{j^*\}} t_j \leq (2 - \frac{1}{m}) L^*$
  - Show $L_i \geq 2L^* - \frac{1}{m}$ with an example:
    - Create $m(m-1)$ jobs of length 1, then 1 job of length m
    - GS distributes $m - 1$ jobs on each of m machines
    - Adding the last job results in $L = 2m - 1$
    - The optimal solution is $L^* = m$ (put m jobs on $m - 1$ machines, last job on 1 machine)
    - Thus $L = (2 - \frac{1}{L^*}) L^* = (2 - \frac{1}{m}) L^*$

Try an alternative GS that sorts jobs in non-increasing order of processing time, $\mathcal{O}(n \log(n + m))$. Note that

- If machine i has the max load and 1 job, solution is optimal
- If there are $> m$ jobs, then $L^* \geq 2t_{m+1}$
  - Since jobs are sorted by time, $t_1, \dots, t_{m+1} \geq t_{m+1}$
  - At least two such jobs go into the same machine, so that machine has $L_i \geq 2t_{m+1}$

We claim $\text{ALG}(I)$ is a 1.5-approximation

- Proceed the same way as the previous proof.
- If i has ≥ 2 jobs, the first m jobs are already filled, so $t_{j^*} \leq t_{m+1}$
- Then $L = L_i = (L_i - t_{j^*}) + t_{j^*} \leq L^* + t_{m+1} \leq L^* + \frac{L^*}{2} \leq 1.5L^*$
- This isn't tight: it turns out this is a $(\frac{4}{3} - \frac{1}{3m})$-approximation – we can show the lower bound by example
  - Create jobs of length $m, \dots, 2m - 1$ twice, plus one job of length m
  - GS distributes $[(2m-1) + (m), (2m-2) + (m+1), \dots]$ so every job has $3m - 1$
  - Adding the last job results in $L = 4m - 1$
  - The optimal solution is $L^* = 3m$ (put $[(2m-1) + (m+1), (2m-2) + (m+2), \dots]$, add $m + m$ plus the last job into the last machine)
  - Thus $L = \frac{4m-1}{3m} L^* = (\frac{4}{3} - \frac{1}{3m}) L^*$

eg. Unweighted vertex cover – find a minimally-sized vertex cover S (vertices that touch every edge)

We proved this problem is NP-complete – consider two greedy algorithms

- Repeatedly take an edge $(u, v) \in E$, add u, v to S, and delete all edges touching u or v
  - The edges we pick, M, form a matching / independent edge set
  - Any vertex cover S has ≥ 1 endpoint of each edge in any matching M, thus $|S| \geq |M|$
  - GS finds a vertex cover of size $2|M| \leq 2|S| \leq 2|S^*|$, where $S^* =$ min vertex cover
- Repeatedly take vertices that maximize new edges touched
  - Provides a $\mathcal{O}(\log \max_{v \in V} v_{\text{degree}})$ approximation

eg. Weighted vertex cover – given vertex weights w_v, find a vertex cover with minimal weight

The greedy approach won't work – you can prove for any ϵ, exists graph with ratio $\geq \epsilon$
Instead, consider the integer LP where $x_v = \mathbb{I}(v \in S)$

$$\begin{aligned} &\min \sum_{v \in V} w_v x_v \\ &\text{subject to } x_u + x_v \geq 1, \forall (u, v) \in E \\ &\qquad\qquad\quad x_v \in \{0,1\}, \qquad \forall v \in V \end{aligned}$$

To make this a normal LP, set $x_v \geq 0$

- Suppose we minimize $\mathrm{c}^{\mathrm{T}}\mathrm{x}$ – the new LP has a larger feasible space, so $\mathrm{ARG}(I) \leq \mathrm{OPT}(I)$
- Let $\mathrm{x}^*_{\mathrm{LP}}, \mathrm{x}^*_{\mathrm{ILP}}$ be some optimal LP and ILP, then $\mathrm{c}^{\mathrm{T}}\mathrm{x}^*_{\mathrm{LP}} \leq \mathrm{c}^{\mathrm{T}}\mathrm{x}^*_{\mathrm{ILP}}$
- Round $\mathrm{x}^*_{\mathrm{LP}}$ to $\hat{\mathrm{x}}$, where $\hat{x}_v = \mathbb{I}(x^*_v \geq 0.5)$ – the solution is still feasible
  - For edge $(u, v) \in E$, by pigeonhole principle, ≥ 1 of $x^*_u, x^*_v \geq 0.5$ and will be rounded up
  - In the worst case, $x^*_u = x^*_v = 0.5$ and we round both, doubling the variable
- Thus, $\widehat{\mathrm{ARG}}(I) = \sum_{v \in V} w_v \hat{x}_v \leq 2 \sum_{v \in V} w_v x^*_v = 2\mathrm{ARG}(I) \leq 2\mathrm{OPT}(I)$

Local Search: Start with an initial feasible solution S, repeatedly move to a better solution in the local neighbourhood

- eg. Network flow – start at 0 flow, check all flows obtainable by augmenting the residual, increase flow
- eg. LP – find any vertex, go to a neighbour vertex with better objective value
- Runs the risk of getting "stuck" at local maxima – we need to bound worst-case ratio between global maxima and worst local maxima

eg. Max-cut – Find partition (A, B) of an undirected graph that maximizes the # of edges going through the cut.

Local search: initialize (A, B) randomly; if $\exists v \in V$ where moving v between A/B improves $|\{e \text{ across } A\}|$, do so.

- Algorithm stops in $\leq m$ iterations
- When algorithm stops, $\forall v \in V, \sum_{(\ldots,v) \text{ across } A} v \geq \sum_{(\ldots,v) \text{ within } A} v$
- This implies at least half of edges go through the cut, so $|A| \geq \frac{|E|}{2} \geq \frac{|A^*|}{2}$

eg. Exact Max-k-SAT – Given an exact k-SAT formula with clause weights w_i, find a truth assignment τ maximizing the total weight of clauses satisfied $W(\tau) = \sum_{i=1}^{m} C_i w_i$.

Define $N_d(\tau)$ = set of truth assignments differing from τ in $\leq d$ variables
For $d = \mathcal{O}(n)$, local search is a $\frac{2}{3}$ approximation
For $d = \Omega(n)$, local search has exponential neighbourhood size, but is a $\leq \frac{4}{5}$ approximation at $d < \frac{n}{2}$

Define S_k = set of clauses where k literals are true under τ, where $W(S_k)$ is the clause's total weight
???????? I dunno lol