FFT & rFFT

- Linear Combination

  $a = \text{randn}(1, 100)$

  $b = \text{randn}(1, 100)$

  $ab = [a; b]$

  $c = [3, -5]$

  $(c \star ab').\text{/diag}(ab \star ab')'$

  $c \star ab' \star \text{inv}(ab \star ab') = [3, -5]$

- Cosine Basis

  $ab = [\cos((.5{:}\,100)/100 \star \text{pi} \star 3); \cos((.5{:}\,100)/100 \star \text{pi} \star 4)]$

  $c = [3, -5] \star ab$

  $(c \cdot ab').\text{/diag}(ab \cdot ab')'$ recovers the coefficients exactly. Why? Because cosine fucntions are orthogonal to each other: the matrix $ab \cdot ab'$ only has entries on its diagonal; vector x, y are orthogonal if their dot product is 0, thus $\text{sum}(a. \star b) \approx 0$, the frequency number has to be between 0 and 99.

- Cosine Transform of an Image

  - *inverse cosine transform*: is the process of expressing a signal interms of cosines for our vector c we computed as $(c \star M').\text{/diag}(M \star M')$,

  - *cosine transform*: is the process of taking coefficients and computing the corresponding linear combination of cosines, which we computed by just multiplying by M.

- Fourier Transforms

  i. Pseudocode for FFT

  Input: $(a_0, a_1, ..., a_{n-1})$ coefficients of a polynomial f of degree $<n$, $n = 2p = 2^k$ (power of 2, assume n' is pad to $n = 2^k$)

  $$\text{let} \begin{cases} f^{(0)} \text{ be the polynomial with coefficients } (a_0, a_2, a_4, ...) \\ f^{(1)} \text{ be the polynomial with coefficients } (a_1, a_3, a_5, ...) \end{cases}$$

  $$\begin{cases} \text{recursively find } \{f^{(0)}(y)\text{: } y \text{ is } p^{\text{th}} \text{ complex root of } 1\} \\ \text{recursively find } \{f^{(1)}(y)\text{: } y \text{ is } p^{\text{th}} \text{ complex root of } 1\} \end{cases}$$

for each j, let $x_j = e^{2\pi ij/n}$, $y \leftarrow x_j^2$, $y = \begin{cases} e^{2\pi ij/P}, & \text{if } j \leqslant P-1 \\ e^{2\pi i(j-P)/P}, & \text{if } j \geqslant P \end{cases}$

$f(x_j) \leftarrow f^{(0)}(y) + x_j f^{(1)}(y)$

Output: $\{f(x_j): 0 \leqslant j \leqslant n-1\}$

ii. Theorem

This is an $O(n \log n)$ algorithm.

iii. Matrix

$$\begin{pmatrix} y_0 \\ y_1 \\ \dots \\ \dots \\ y_j \\ \dots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^k & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & \dots & \dots & x_1^{n-1} \\ \dots & & & & & & \\ \dots & & & & & & \\ 1 & x_j & x_j^2 & \dots & e^{2\pi i(jk)/n} & \dots & x_j^{n-1} \\ \dots & & & & & & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & & \dots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ \dots \\ a_j \\ \dots \\ a_{n-1} \end{pmatrix}$$

- Inverse FFT

i. Given: the images of the n^{th} roots of 1 by an unknown polynomial of degree $\leqslant n-1$

ii. Output: the coefficients of the polynomial

iii. Call it $p(x)$, the unknown coefficients: a_0, a_1, ..., a_{n-1}, $p(x) = a_0 + a_1 x + \dots + a_{n-1}x^{n-1}$

iv. Call the n^{th} roots of unity: $x_0, x_1, \dots, x_{n-1}$, $x_j = e^{2\pi ij/n}$

v. Call their images: $y_0, y_1, \dots, y_{n-1}$

vi. We want $a_0, a_1, \dots, a_{n-1}$, this is a linear system of equation

vii. We just need to invert the above matrix, let $z = e^{2\pi i/n}$

$$M = \begin{pmatrix} 1 & z^0 & z^0 & \dots & z^{0(k)} & \dots & z^{0(n-1)} \\ \dots & & & & & & \\ 1 & z^j & z^{2j} & \dots & z^{jk} & \dots & z^{j(n-1)} \\ \dots & & & & & & \\ 1 & z^{n-1} & z^{2(n-1)} & \dots & \dots & \dots & z^{(n-1)^2} \end{pmatrix}$$

the Van der Mondes matrix has a non-zero determinant, inverse matrix is:

$$M^{-1} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 & \dots & 1 \\ 1 & z^{-1} & z^{-2} & \dots & & & \\ 1 & z^{-2} & z^{-4} & \dots & & & \\ 1 & \dots & \dots & z^{-jk} & & & \\ \dots & & & & & & \\ 1 & & & & & & \end{pmatrix} \cdot \frac{1}{n}$$

viii. $(a_j) = M^{-1}(y_j)$

- Relation between FFT and Convolution

  $p(x) = \sum_{j=0}^{n-1} a_j x^j$, $q(x) = \sum_{i=0}^{n-1} b_i x^i$, $C_k = \sum_{j+i=k} a_j b_i$.

  $(a_0 + a_1 x)(b_0 + b_1 x) = \underbrace{a_0 b_0}_{C_0} + \underbrace{(a_0 b_1 + a_1 b_0)}_{C_1} x + \underbrace{a_1 b_1 x^2}_{C_2}$

  $a^T = (a_0, \dots, a_{n-1})$, $b^T = (b_0, \dots, b_{n-1})$, $a \otimes b = C_{2n} = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a) \cdot \mathrm{DFT}_{2n}(b))$

- To multiply $A(x) = a_0 + a_1 x + \dots + a_{n-1} x^{n-1}$ by $B(x) = b_0 + b_1 x + \dots + b_{n-1} x^{n-1}$:

  a) find $(x_i, A(x_i))$ and $(x_i, B(x_i))$ for $2n-1$ different values of x_i — $O(n \log n)$ with *FFT*

  b) for all i compute $p(x_i) \cdot q(x_i)$ — $O(n)$

  c) $pq(x)$ is the polynomial with points $(x_i, p(x_i) \cdot q(x_i))$ of $deg \leqslant 2n-2$ — $O(n \log n)$ with *iFFT*

- Application: Big-Multiplication

```
BIG-MULT(a, b)
 1  na=length(a); nb=length(b);
 2  a=[a zeros(1, nb-1)]; b=[b zeros(1, na-1)];
 3  c=ifft(fft(a).*fft(b));
 4  ...
```

Dynamic Programming

- *Number of paths in a graph*: If we multiply by M repeatedly, this corresponds to longer paths: the lth entry of the vector $e_i \cdot M^k$ records the number of paths of length k in G going from i to l. *Recursive representation of paths in a graph*:

PATHS(i,k,l):
return: $\sum_j \text{PATHS}(i,k-1,j)\cdot M(j,l)$

Minimum cost paths in a graph:

PATHS(i,k,l):
return: $\min_j \text{PATHS}(i,k-1,j)+M(j,l)$

- Egg Dropping Problem

  Let $f(t,n)$ be the most floors given t trials, n eggs (i.e. at most t depth, at most n left moves, $f(t,\ n) =$ most number of nodes to add to tree). $f(t,n)=1+f(t-1,n-1)+f(t-1,n)$

- Edit Distance

  The recurrence for editDistance from the problem statement is:

$$S(i,j)=\begin{cases} S(i-1,j-1) & \text{if } A[i]=B[j] \\ \min\left(S(i-1,j),S(i,j-1),S(i-1,j-1)\right)+1 & \text{if } A[i]\neq B[j] \end{cases}$$

  Recall the *meaning* of $S(i,\ j)$: it is the edit distance between the first i characters of the first string A and the first j characters of the second string B. Thus we want to compute $S(A.\text{length},\ B.\text{length})$, which we do via dynamic programming, filling in all entries $S(i,\ j)$ for i between 0 and A.length and j between 0 and B.length, leading to a 2-dimensional array with dimensions $(A.\text{length}+1)\times(B.\text{length}+1)$

  The base cases for our algortihm encode the fact that the edit distance between the empty string and a string of length i (or j) is i (or j respectively):

$$\begin{aligned} S(i,0) &= i, & \forall i, 0\leqslant i\leqslant A.\text{length} \\ S(0,j) &= j, & \forall j, 0\leqslant j\leqslant B.\text{length} \end{aligned}$$

- Bellman-Ford Algorithm

```
BELLMAN-FORD()
1  A ← 2D array (indexed by i and v)
2  A[0, s] = 0
3  A[0, v] = +∞, ∀v, v ≠ s
4  for i ← 1 to n − 1:
5      for each v ∈ V do:
6          A[i, v] = min(A[i − 1, v], min(A[i − 1, w] + c_wv, ∀(w, v), (w, v) ∈
```

- Knapsack Problem

```
KNAPSACK(W)
1  A ← 2D array (indexed by i and v)
2  initialize A[0, x] = 0, ∀x, x ∈ {0, 1, 2, ..., W}
3  for i = 1, 2, ..., n:
4      for x = 0, 1, 2, ..., W:
5          A[i, x] ← max {A[i − 1, x], A[i − 1, x − w_i] + v_i}    /*x − w_i ⩾ 0 *
6  return A[n, w]
```

- Optimal Binary Search Tree

```
OPTIMAL-BST()
1  A ← 2D array
2  for s ← 0 to n − 1 do: /* s represents (j − i)*/
3      for i ← 0 to n /* i represents contiguous interval */
4          A[i, i + s] ← min_{r=1}^{i+s} {∑_{k=i}^{i+s} p_k + A[i, r − 1] + A[r + 1, i + s]}
5                                  /* as 0 if 1st index > 2nd index*/
6  return A[1, n]
```

- Longest Palindromic Subsequences

We describe a dynamic programming approach for coy mputing the length of the longest palindromic subsequence in a string of characters. We will define a recurrence in terms of $T(i, j)$, representing the *length of the longest palindromic subsequence consisting of characters from i to j inclusive*. We now derive a recurrence relation for T.

There are two cases to the recurrence. Either the first and last character of the substring from i to j match, or they do not:

i. If they match, then the length of the longest palindromic subsequence from i to j must be two more than the length of the longest palindromic subsequence from $i+1$ to $j-1$.

ii. If they do not match, then no palindromic subsequence can both start with the ith character and end with the jth character, so the longest palindromic subsequence between i and j must be *either* the longest palindromic subsequence between i and $j-1$, or the longest palindromic subsequence between $i+1$ and j.

Written explicitly, the recurrence is:

$$T(i,j) = \begin{cases} T(i+1, j-1) + 2 & \text{if } s[i] = s[j] \\ \max\left(T(i+1, j), T(i, j-1)\right) & \text{if } s[i] \neq s[j] \end{cases}$$

Since the recurrence only makes use of entries with greater first index, or smaller second index, we can fill in the table with two nested loops, where one loop decreases the first index and the other loop increases the second index.

It is fairly clear that the following two base cases are enough to define the rest of the values in T:

- String of length 1: for all i between 0 and $n-1$, let $T(i,j)=1$.
- String of length 0: for all i between 1 and $n-1$, let $T(i,i-1)=0$.

The following is the pseudocode for our algorithm, which accepts as input a string s and outputs the length of the longest palindromic subsequence:

```
PALINDROME(s)
 1  Set n ← LENGTH(s)
 2  Create 2-dimensional array T[n][n]
 3  for i ← 0 to n − 1
 4     do T(i, i) = 1
 5  for i ← 0 to n − 1
 6     do T(i, i − 1) = 0
 7  for i ← n − 1 down to 0
 8     do for j ← i + 1 to n − 1
 9        do if s(i) = s(j)
10           then T(i, j) ← T(i + 1, j − 1) + 2
11           else T(i, j) ← max(T(i + 1, j), T(i, j − 1))
12  Return T(0, n − 1)
```

Proof Of Correctness

As we derived the pseudocode above, we introduced all the elements needed for a proof of correctness. We need only assemble the pieces:

(1)We argued above that the recurrence for $T(i, j)$ correctly expresses the length of the longest palindromic subsequence between location i and j in terms of values that have already been computed. (2)And finally, the base cases, strings of length 0 and 1, clearly have longest palindromic subsequences of lengths 0 and 1 respectively. (3)Therefore, induction on the number of times line 9 in the pseudocode is executed yields a proof of correctness of the values of the table T, and in particular, guarantees that the answer $T(0,n-1)$ returns the length of the longest palindromic subsequence of the entire string.

Running Time Analysis

We fill half of a table that contains $O(n^2)$ elements. In filling each element, we do constant work. Thus, the runtime of our algorithm is $O(n^2)$.

- ***Maximum Sum*** of a Triangle

  Let T be a triangle of numbers with r rows, where $T[i, j]$ is the number in the ith row of T that is j from the left, starting from 1 (e.g., $T[3, 2]$ in the example triangle is 4). We define an algorithm defined by the following recurrence relation:

$$S[i,j] = \begin{cases} T[i,j] & \text{if } i = r \\ T[i,j] + \max\left(T[i+1,j], T[i+1,j+1]\right) & \text{if } i < r \end{cases}$$

  where $1 \leqslant i \leqslant r, 1 \leqslant j \leqslant i$ and $S[i, j]$ is the maximum sum from row r of T to $T[i, j]$.

  $S[i, j]$ is an $r \times r$ table. We will fill only the bottom half of the table (we only need $i \geqslant j$), starting at the bottom row and working across rows to the top. The maximum sum in T will be stored at $S[1, 1]$.

```
MAX-SUM(T)
 1  r ← T.numRows
 2  S ← n × n matrix
 3  for j ← 1 to r
 4     do
 5        S[r, j] ← T[r, j]
 6  for i ← r − 1 to 1 do
 7     for j ← 1 to i do
 8        S[i, j] ← T[i, j] + max(T[i + 1, j], T[i + 1, j + 1])
 9  Retrun S[1, 1]
```

  Proof Of Correctness

  We can prove that the algorithm is correct by showing that the recurrence relation works and each $S[i, j]$ is the maximum sum from row r of T to $T[i, j]$.

  - Inductive Hypothesis: The algorithm returns the maximum sum for a triangle with $r = k$ rows.

  - Base Case: First, the base case ($r = 1$) is true. If we have a triangle with only one row, then it can only have one entry, and it is clear that the maximum sum in the tree is exactly the value of that entry.

– General Case: We want to show that the algorithm returns the correct answer when $r = k+1$. Starting from the top of the triangle , the path followed to obtain the maximum sum must pass through one of the numbers in the second row of T. Both of these taken individually are the top row of triangles with k rows , so we know from the induction hypothesis that $S[2, j]$ stores the maximum sum from the base of T to the numbers in the second row. Since the path to the top row is obtained by adding this sum to the value stored in the first row of T, it is clear that the maximum sum for T will be obtained by adding $T[1, 1]$ to the maximum of the sums stored in the second row of T, which is exactly what is calculated by the recurrence relation. Therefore, our algorithm is correct for $r = k+1$ and therefore for all r.

Running Time

During the execution of this algorithm, we consider each number a constant number of times. Therefore, the algorithm runs in $O(n)$ where n is the number of elements in the triangle.

- ***MinCost*** of Cutting Wood

Input: a sequence of cuts $A = \{a_1, a_2, ..., a_n\}$, where a_i is the distance from the left edge of the piece of wood, and L, the length of the piece of wood.

For the recurrence relation, redefine $A = \{a_1, a_2, ..., a_n, a_{n+1}\} = \{a_1, a_2, ..., a_n, L\}$ such that in the recurrence relation, $a_{n+1} = L$. The recurrence relation for this problem can be described as

$$C[i, j] = \begin{cases} 0 & \text{if } i = j \text{ or } i = j-1 \\ a_j - a_i + \min_{i<k<j}(C[i, j] + C[k, j]) & \text{if } i \neq j \text{ and } i \neq j-1 \end{cases}$$

where $C[i, j]$ is the minimum cost of cutting the wood between cuts a_i and a_j and $i \leftarrow 0$ to n and $j \leftarrow 1$ to $n+1$.

$C[i, j]$ is a table in which only the upper half is filled (we only need $i \leqslant j$ since we are looking at the cost of cuttin between positions a_i and a_j, so all other entries would be redundant.) Each entry $C[i, j]$ holds the minimum cost of cutting the wood between positions a_i and a_j, and the table is filled diagonally, from the center diagonal to the upper-right corner.

MIN-COST($\{a_1, a_2, ..., a_n\}, L$)

```
1  A ← {a_1, a_2, ..., a_n, L}
2  for i ← 1 to n
3     do for j ← 0 to n + 1 − i
4        do if i = 1
5           then C[j, j + i] ← 0
6           else C[j, j + i] ← a_{j+1} − a_j + min_{j<k<j+i} (C[j, k] + C[k, j +
7  return C[0, n + 1]
```

Proof Of Correctness

We can prove that the algorithm is correct by showing that the recurrence relation works and each $C[i, j]$ is the minimum cost of cutting between positions a_i and a_j, since the algorithm follows the recurrence relation defined above.

- Base Case: First, the base cases are true. In the base case where $i = j$, it is obvious that there is no cut betwen a position and itself. In the base case where $i = j - 1$, there are no cuts between positions a_i and a_j, hence the cost is again 0.

- Now take all other cases in which there is at least one cut between positions a_i and a_j. Suppose the minimum cost $C[i, j]$ for making all cuts between a_i and a_j is obtained by making the next cut at some position a'_k, $i < k' < j$. Then $C[i, j] = a_j - a_i + C[i, k'] + C[k', j]$. Because $a_j - a_i$ is constant for any given i, j (the length of a piece of wood does not change regardless of how it will be cut), $C[i, k'] + C[k', j]$ is the minimum of all $C[i, k] + C[k, j], \forall i, i < k < j$, where a_k is the next cut, and the recurrence relation(and hence the algorithm) holds.

Running Time

The recurrence relation fills in the upper triangle of a table with dimensions n by n. There are thus $n^2/2$ or $O(n^2)$ entries in the table. For every entry $C[i, j]$, we perform on the order of $O(n)$ operations by comparing the sum of the entries $C[i, k] + C[k, j]$ for all $i < k < j$. This is a linear number of operations on the order of n since in the worst case, we have $0 < k < n + 1$ and have to sum the entries $C[0, 1] + C[1, n + 1]$ through $C[0, n] + C[n, n + 1]$. The total runtime is thus $O(n^3)$.

Divide & Conquer

- ***Medium-Find***

Runtime: depends on how "lucky" we are → "unlucky" if x is always the largest or smallest elem of L (e.g., L is sorted). It would be $O(n^2)$. (1)So instead of choosing x to be the first element, let it be a random element. (2)Assume fixed input, prove it run in $O(n)$ time. *Claim*: runtime depends only on the size of L. Let $T(n)$ b the time the algo takes on average, suppose that $|L_1|=5, |L_2|=n-5-1$. Then we either need to do $T(5)$ or $T(n-5-1)$. In the worst case, it will be $\max(T(5), T(n-5-1))$.

Let $R(x)$ be the rank of x in L. Then $|L_1|=R(x)-1, |L_2|=n-R(x)$ then the time will be at most max $(T(|L_1|), T(|L_2|))$. When $|L_1| = |L_2|$, then $T(n)=T(n/2)+n$. If $n/4 \leqslant R(x) \leqslant 3n/4$, then $|L_1| \leqslant 3n/4$, $|L_2| \leqslant 3n/4 \Rightarrow$ $T(n) \leqslant T(3n/4) + n$. The probability of this happening is $1/2$, since the distribution of $R(x)$ is uniform ⇒ Takes a constant number of "coin flips" (choices of random numer) to get to this case ⇒ takes time $O(n)$ [expected to take this amount of time].

- ***Select a pivot deterministically***

  (1) divide the list into groups of 5 elements; (2) for each group, find its mediam →$O(1)$ for each ⇒$O(n)$ total; (3) find the median of these $n/5$ elements recursively and use it as your pivot.

  Why does this guarantee a good pivot? If there are n elements total, then the pivot is greater than at least $3n/10$ elements, and less than at least $3n/10$ elements ⇒ $3n/10 \leqslant R(\text{pivot}) \leqslant 7n/10 \Rightarrow T(n) \leqslant T(7n/10) + T(n/5) + n$, base case 1.

- ***FindMax*** value increasing from the left, descending to the right

  Given a location i in the 1-indexed array A of length n, we can clearly tell if i is to the left of the max, to the right of the max, or is the max, via the following three-way test:

$$\begin{cases} i < n \text{ and } A[i] < A[i+1] & \rightarrow & \texttt{max} \text{ is to the right of } i \\ i > 1 \text{ and } A[i] < A[i-1] & \rightarrow & \texttt{max} \text{ is to the left of } i \\ \text{else} & \rightarrow & \texttt{max} \text{ is at } i \end{cases}$$

  Knowing whether the desired location is to the left, to the right, or on the current location is exactly the subroutine used in the binary search algorithm, which will thus solve the problem in $O(\log n)$ time (since each test can be done in constant time).

- ***BigWin*** $O(n \log n)$ Solution

Heuristic: Starting at the center element, c, we find the maximum sum, L, of consecutive elements to the left of c that includes c. Similarly, we find the maximum sum, R, of consecutive elements to the right of c that includes c. Then, *BigMiddleWin* should return $L + R - c$.

Algorithm: Consider the center element, c, in the list. Either the maximum sum is completely contained in the elements to the left of c, completely contained in the elements to the right of c, or passes through c. We have already seen how to solve the last case with *BigMiddleWin*. We can then consider an algorithm, *BigWin*, which takes as input a list, l. It returns the max of *BigMiddleWin* of l, *BigWin* of the left half of l, and *BigWin* of the right half of l.

Correctness: We can assume that *BigMiddleWin* will return the correct maximum of a given list. Any contiguous subsequence from a list l either contains the middle element or it doesn't. In the case that it does, the maximum of all such paths will be returned by *BigMiddleWin*. Since the sequence is contiguous, if it does not contain the middle element then all elements must fall in the left half, or they must fall in the right half. These can both be treated as independent lists (since they share no elements), and so we can solve them independently. *Thus, the maximum total sum will be the max of the sum that considers only the left half, the sum that considers only the right half, and the sum that considers the whole list together*.

Runtime: The algorithm leads to a recurrence of the form $T(n)=2T(n/2)+O(n)$. We have solved this recurrence before, and know it is $O(n \log n)$.

Greedy Algorithm

- ***Cloest Hotel***

  Each morning, Bilbo will choose for his next night the hotel closest to his destination that he can reach that day, or his destination if he can reach his destination that day.

  We claim, by induction, that for each $i \geqslant 0$, under this strategy at the ith night Bilbo will be as close as possible to his destination. If at night i Bilbo is at location x, which is the greatest possible location he could be along his route for night i, and on the next day he travels to hotel at greatest location less than or equal to $x+20$, then clearly at night $i+1$ he will be as far as possible along his route for night $i+1$, proving the induction. Thus Bilbo greedy strategy will get him as far as possible each day, leading him to finish his journey in the least number of days.

- ***Attend Events***

  At the moment each event starts, choose a person from your dorm room to go to that event.

  This strategy can only fail if there is no one in your dorm room when an event starts, meaning that all your k friends are already at k events. These k events and the just-starting event make $k+1$ events. Since there are $k+1$ events happening simultaneously, there is no way to attend each event with only k people. Thus the only way our strategy fails is if our task is impossible, meaning our algorithm will succeed in all possible cases.

- ***Scheduling Jobs*** PATHS(i,k,l):
  return: $\sum_j$ PATHS$(i,k-1,j)\cdot M(j,l)$

  Claim: algorithm (order jobs according to decreasing ratios w_i/l_i) is always correct.

  Proof: by an *Exchange Argument*, by Contradiction. **Plan**: fix arbitrary input of n jobs, will proceed by contradiction. Let $\sigma =$ greedy schedule, $\sigma^* =$ optimal schedule, will produce schedule even better than σ^*, contradicting purported optimality of σ^*.

  Assume: all w_i/l_i are distinct; [just by renaming jobs] $w_1/l_1 > w_2/l_2 > ... > w_n/l_n$, where job$_1$ has the greatest ratio.

  Thus: greedy schedule sigma is just 1, 2, 3, ..., n. If optimal schedule σ^* is inequivalent to σ, then there are consecutive jobs i, j with $i > j$. [only schedule where indices always go up is $1, 2, ..., n$].

  Thought Experiment: ramifications of the exchange of jobs $\rightarrow$ it has to have this pair of consecutive jobs where the earlier one has the higher index; *Effect of this exchange on the completion time of*: $^{(1)}k$ other than i or j: before the swap, it has to wait for a bunch of ops to complete, including i and j, the amount of time elapsed is the same, thus jobs other than i and j are completely agnostic to the swap; k preceed i and j is the same. $^{(2)}$job i: goes up, now it has to wait for j, goes up by the lenght of j. $^{(3)}$job j: drops, no longer has to wait for job i, goes down precisely the length of job i.

  Cost-Benefit Analysis: *Upshots*: $i > j \Rightarrow \frac{w_i}{l_i} < \frac{w_j}{l_j} \Rightarrow w_i l_j < w_j l_i \Rightarrow$ cost $<$ benefit $\Rightarrow$ swap improves σ^*, contradicts optimality of σ^*. *QED*!

- ***Prim's MST Algorithm***

  PRIM(s)

```
1  X ← {s}
2  T ← EMPTY
3  while X != V:
4      e ← (u, v) /* be the cheapest edge of G with u in X, v not in X
5      T ← T ∪ {e}
6      X ← X ∪ {v}
```

- ***Kruskal's MST Algorithm***

```
KRUSKAL(s)
1  sort edges in increasing order /* bottle-neck in the end O(m log n) *
2  [rename edge 1, 2, ..., m, so that c_1 < c_2 < c_3 ... < c_m]
3  T = EMPTY SET
4  for i = 1 to m do:
5      if T ∪ {i} has no cycle then do: /* originally O(n) without Unio
6          T ← T ∪ {i}
7  return T
```

- ***Union-Find for Kruskal MST***

Raison d'etre: maintain partition of a set of objects. **Motivation**: $\theta(1)$ time for cycle checks.

Find(x): return name of group that x belongs to. **Union(c_i, c_j)**: fuse groups c_i and c_j into a single one.

Idea: when two componenets merge, have smaller one inherit the leader of the larger one; you have to quickly determine which group is larger, so you may maintain a size field for each group. *How many leader pointer updates are now required to restore the invariant in the worst case?* $\boldsymbol{\theta(\log n)}$. **Reason**: you are joining a group at least as big as yours, so the size of the union, the size of your new community, is at least double the size of your previous one.