CMPT231

Lecture 2: ch4-5

Divide and Conquer, Recurrences, Randomised Algorithms

James 1:2-4 (NASB)

Consider it **all joy**, my brethren,
when you encounter various **trials**,
knowing that the **testing of your faith** produces endurance.

And let **endurance** have its perfect result,
so that you may be **perfect and complete**,
lacking in nothing.

James 1:5-8 (NASB)

But if any of you **lacks wisdom**, let him ask of God,
who gives to all **generously** and **without reproach**,
and it will be given to him.

But he must **ask in faith** without any doubting,
for the one who doubts is like the **surf of the sea**,
driven and tossed by the wind.

For that man ought not to expect
that he will receive anything from the Lord,
being a **double-minded** man, **unstable** in all his ways.

Outline for today

- **Divide and conquer** (ch4)
  - **Merge sort**, recursion tree
  - Proof by **induction**
  - Maximum **subarray**
  - Matrix multiply, **Strassen**'s method
  - **Master method** of solving recurrences
- **Probabilistic Analysis** (ch5)
  - **Hiring** problem and analysis
  - **Randomised** algorithms and PRNGs

Divide and conquer

- Insertion sort was **incremental**:
  - At each step, A[1 .. j-1] has been sorted, so
  - Insert A[j] such that A[1 .. j] is now sorted
- Divide and conquer **strategy**:
  - **Split** task up into smaller chunks
  - Small enough chunks can be solved **directly** (base case)
  - **Combine** results and return
- Implement via **recursion** or **loops**
  - Usually, recursion is **easier** to code but **slower** to run

Merge sort

- **Split** array in half
  - If only 1 elt, we're done
- **Recurse** to sort each half
- **Merge** sorted sub-arrays
  - Need to do this efficiently

```
def merge_sort(A, p, r):
  if p >= r: return
  q = floor( (p+r) / 2 )
  merge_sort(A, p, q)
  merge_sort(A, q+1, r)
  merge(A, p, q, r)
```

Efficient merge in Θ(n)

- Assume subarrays are **sorted**:
  - A[p .. q] and A[q+1 .. r], with $p \leq q < r$
- Make temporary **copies** of each sub-array
  - Append an "∞" **marker** item to end of each copy
- **Step** through the sub-arrays, using two indices (i,j):
  - Copy **smaller** element into main array
    - and **advance** pointer in that sub-array
- **Complexity**: $\Theta(n)$

```
Main : [ A, C, D, E, H, J,  ,  ,  ,  ,  ,  ]
L: [ C, E, H, *K, P, R, inf ]  ||  R: [ A, D, J, *L, N, T, inf ]
```

Merge step: in pseudocode

```
def merge(A, p, q, r):
  ( n1, n2 ) = ( q-p+1, r-q )

   # lengths
  new arrays: L[ 1 .. n1+1 ], R[ 1 .. n2+1 ]

  for i in 1 .. n1: L[ i ] = A[ p+i-1 ]
 # copy
  for j in 1 .. n2: R[ j ] = A[ q+j ]
  ( L[ n1+1 ], R[ n2+1 ] ) = ( inf, inf )       # sentinel

  ( i, j ) = ( 1, 1 )
  for k in p .. r:
    if L[ i ] <= R[ j ]:
```

Complexity of merge sort

- **Recurrence** relation: **base** case + **inductive** step
  - **Base** case: if $n = 1$, then $T(n) = \Theta(1)$
  - **Inductive** step: if $n > 1$, then $T(n) = 2T(n/2) + \Theta(n)$
    - **Sort** 2 halves of size $n/2$, then **merge** in $\Theta(n)$
- How to **solve** this recurrence?
  - Function **call diagram** looks like binary tree
  - Each **level** L has 2^L recursive calls
  - Each call performs 2^{-L} work in the merge step

Recurrence tree

- Total work at each level is $\Theta(n)$
  - Total number of levels is $\lg(n)$
  - $\Rightarrow$ Total complexity: $\Theta(n \lg(n))$
- This is **not** a formal proof!
  - A **guess** that we can prove by **induction**

Outline for today

- Divide and conquer (ch4)
  - Merge sort, recursion tree
  - **Proof by induction**
  - Maximum subarray
  - Matrix multiply, Strassen's method
  - Master method of solving recurrences
- Probabilistic Analysis (ch5)
  - Hiring problem and analysis
  - Randomised algorithms and PRNGs

Mathematical induction

- **Deduction**: general principles ⇒ specific case
- **Induction**: representative case ⇒ general rule
- Proof by induction needs two **axioms**:
  - **Base case**: starting point, e.g., at $n = 1$
  - **Inductive step**: assuming the rule holds at n,
    - **prove** it also holds at $n+1$
    - equivalently, assume true $\forall\ m < n$, and prove for n
- This proves the rule for **all** (positive) n

Example of inductive proof

- Recall **Gauss**‘ formula: $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$
- We can **prove** it by induction:
- Prove **base case** (n = 1): 1 = (1)(1+1)/2
- Prove **inductive step**:
  - **Assume**: $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$
  - **Prove**: $\sum_{j=1}^{n+1} j = \frac{(n+1)(n+2)}{2}$

Inductive step of Gauss' formula

- $\sum_{j=1}^{n+1} j = \left(\sum_{j=1}^{n} j\right) + (n+1)$
- $= \frac{n(n+1)}{2} + (n+1)$ (by **inductive hypothesis**)
- $= \frac{n^2+n}{2} + (n+1)$
- $= \frac{n^2+n+2n+2}{2}$
- $= \frac{n^2+3n+2}{2}$
- $= \frac{(n+1)(n+2)}{2}$, **proving** the inductive step

Inductive proof for merge sort

- **Recurrence**: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$, with T(1) = Θ(1)
- **Guess** (from recursion tree): T(n) = Θ(n lg n)
- Prove **base case**: T(1) = Θ(1 lg 1) = Θ(1)
- **Inductive hypothesis**: assume ∃ c_1, c_2, n_0: ∀ n_0 < m < n, $c_1 m \lg m \le T(m) \le c_2 m \lg m$
- **Inductive step**: with the **same** constants c_1, c_2, we want to **prove** $c_1 n \lg n \le T(n) \le c_2 n \lg n$

Inductive step for merge sort

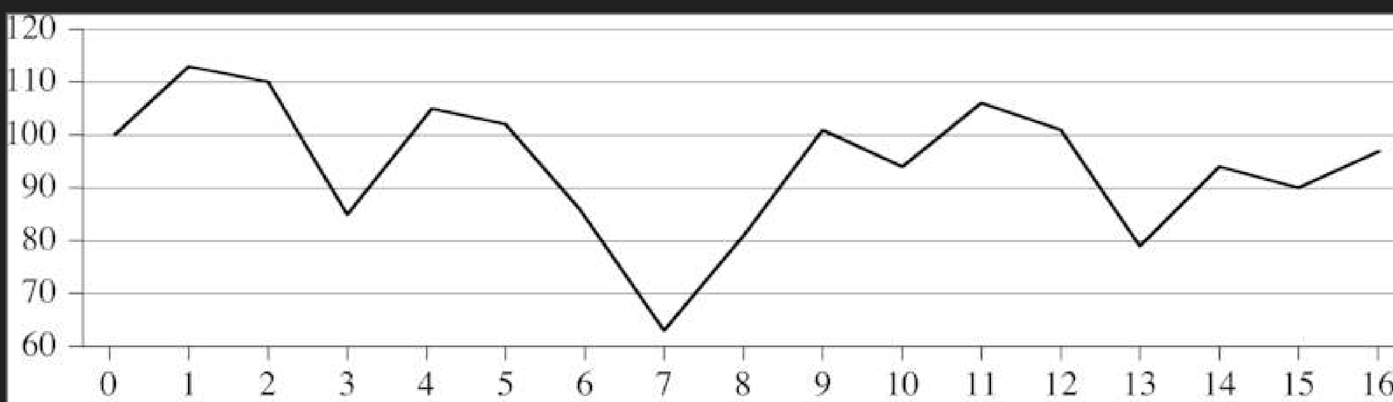
- Use **recurrence** and defn of Θ(n): ∃ c_3, c_4:
  $2T\left(\frac{n}{2}\right) + c_3 n \le T(n) \le 2T\left(\frac{n}{2}\right) + c_4 n$
- Apply **inductive hypothesis** with m = n/2:
  $2c_1\left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right) + c_3 n \le T(n) \ \le 2c_2\left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right) + c_4 n$
- $\Rightarrow c_1 n(\lg n - \lg 2) + c_3 n \le T(n) \ \le c_2 n(\lg n - \lg 2) + c_4 n$
- $\Rightarrow c_1 n \lg n + (c_3 - c_1) n \le T(n) \ \le c_2 n \lg n + (c_4 - c_2) n$
- $\Rightarrow c_1 n \lg n \le T(n) \le c_2 n \lg n$
- **Last step** possible by choosing $c_1 < c_3$ and $c_2 > c_4$

Outline for today

- Divide and conquer (ch4)
  - Merge sort, recursion tree
  - Proof by induction
  - **Maximum subarray**
  - Matrix multiply, Strassen's method
  - Master method of solving recurrences
- Probabilistic Analysis (ch5)
  - Hiring problem and analysis
  - Randomised algorithms and PRNGs

Maximum subarray

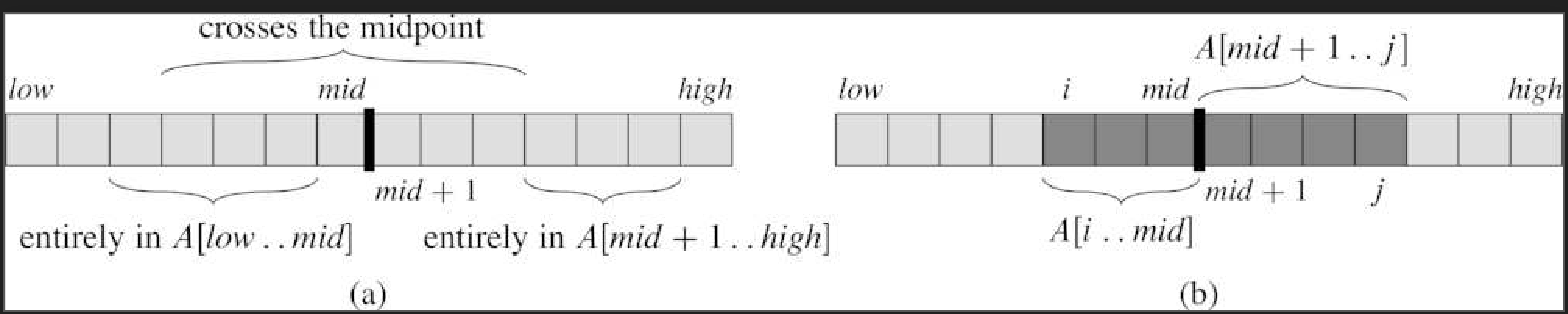
- **Input**: array A[1 .. n] of numbers (could be negative)
- **Output**: indices (i,j) to maximise sum(A[i .. j])
  - e.g., input daily change in **stock** price:
    - find optimal time to **buy** (i) and **sell** (j)
- **Exhaustive** check of all (i,j): $\Theta(n^2)$



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

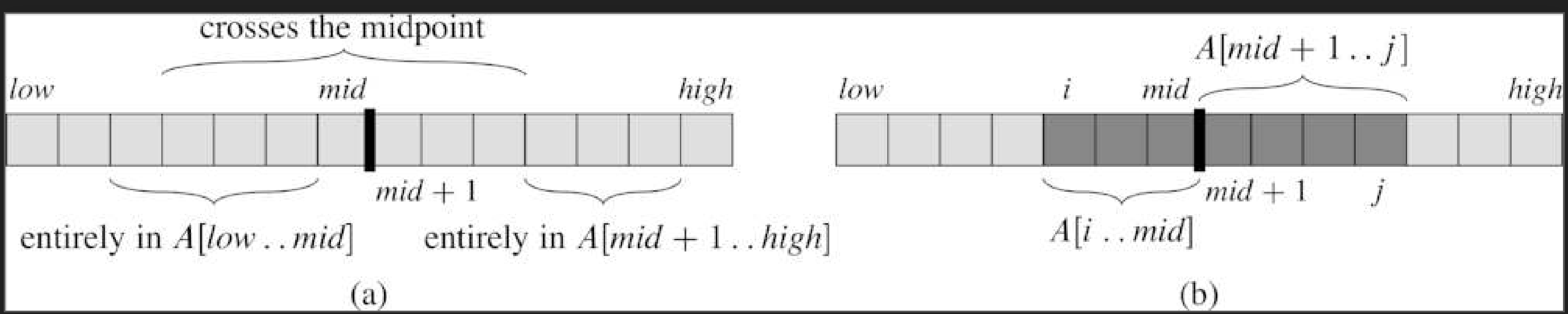
Max subarray: algorithm

- **Split** array in half
- **Recursively** solve each half
  - (what's the **base** case?)
- Find the max subarray which **spans** the midpoint
  - Do this in $\Theta(n)$
- Choose **best** out of 3 options (left, right, span) and return



Span midpoint

- Find the maximum subarray that **spans** the midpoint
- **Decrement** i down from the **midpoint** to the **low** end
  - Maximise sum(A[i .. mid])
- **Increment** j up from `mid+1` to the **high** end
  - Maximise sum(A[mid+1 .. j])
- Total time is only **linear** in n



Max subarray: complexity

```
def max_subarray(A, low, mid, high):
  split_array()


 # O(1)
  max_subarray( left_half )
     # T(n/2)
  max_subarray( right_half )
    # T(n/2)
  midpt_max_subarray()

  # Theta(n)
  return best_of_3()

    # O(1)
```

- **Recurrence**: $T(n) = 2T(n/2) + \Theta(n)$
  - **Base** case: $T(1) = O(1)$
- Same as merge sort: **solution** is $T(n) = \Theta(n \lg n)$

Programming Joke

- There's always a way to **shorten** a program by one line.
  - But, there's also always one more **bug**.
  - $\Rightarrow$ By **induction**, any program can be shortened to a **single line**, which **doesn't work**.

Outline for today

- Divide and conquer (ch4)
  - Merge sort, recursion tree
  - Proof by induction
  - Maximum subarray
  - **Matrix multiply, Strassen's method**
  - Master method of solving recurrences
- Probabilistic Analysis (ch5)
  - Hiring problem and analysis
  - Randomised algorithms and PRNGs

Matrix multiply

- **Input**: two n x n matrices A[i,j] and B[i,j]
- **Output**: C = A ∗ B, where $C[i,j] = \sum_{k=1}^{n} A[i,k]B[k,j]$
- e.g., $\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$
- **Simplest** method:

```
def mult(A, B, n):
  for i in 1 .. n:
    for j in 1 .. n:
      for k in 1 .. n:
C[i, j] += A[i, k] * B[k, j]
  return C
```

Complexity? Can we do **better**?

Divide-and-conquer mat mul

- **Split** matrices into **4** parts (assume n a power of 2)
- **Recurse** 8 times to get products of sub-matrices
- Add and **combine** info final result:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

  - $C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$
  - $C_{12} = A_{11} * B_{12} + A_{12} * B_{22}, \ldots$
- What's the **base case**?
- How to **generalise** to n not a power of 2?

Complexity of divide-and-conquer

- **Split**: O(1) by using **indices** rather than copying matrices
- **Recursion**: 8 calls, each of time T(n/2)
- **Combine**: each entry in C needs one add: $\Theta(n^2)$
- So the **recurrence** is: $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$
  - Unfortunately, this resolves to $\Theta(n^3)$
  - **No better** than the simple solution
- What gets us is the 8 **recursive** calls
  - **Strassen**'s idea: save 1 recursive call
  - by spending more on **sums** (which are only

Strassen’s matrix multiply

- 10 **sums** of submatrices: $S_1 = B_{12} - B_{22}$, $S_2 = A_{11} + A_{12}$, $S_3 = A_{21} + A_{22}$, $S_4 = B_{21} - B_{11}$, $S_5 = A_{11} + A_{22}$, $S_6 = B_{11} + B_{22}$, $S_7 = A_{12} - A_{22}$, $S_8 = B_{21} + B_{22}$, $S_9 = A_{11} - A_{21}$, $S_{10} = B_{11} + B_{12}$.
- 7 **recursive** calls: $P_1 = A_{11} * S_1$, $P_2 = S_2 * B_{22}$, $P_3 = S_3 * B_{11}$, $P_4 = A_{22} * S_4$, $P_5 = S_5 * S_6$, $P_6 = S_7 * S_8$, $P_7 = S_9 * S_{10}$.
- 4 **results** via addition: $C_{11} = P_5 + P_4 - P_2 + P_6$, $C_{12} = P_1 + P_2$, $C_{21} = P_3 + P_4$, $C_{22} = P_5 + P_1 - P_3 - P_7$
.

Complexity of Strassen's method

- Even though more **sums** are done, still all $\Theta(n^2)$
  - Doesn't change total **asymptotic** complexity
  - Might not be worth it for **small** n, though
- **Recurrence**: $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$
  - Saved us 1 recursive call!
  - **Solution**: $T(n) = \Theta(n^{\lg 7})$
- This is an example of the **master method**
  - For recurrences of **form** T(n) = a T(n/b) + Θ(f(n))
  - **Compare** f(n) with $n^{\log_b a}$
  - Is more work done in **leaves** of tree or **roots**?

Outline for today

- Divide and conquer (ch4)
  - Merge sort, recursion tree
  - Proof by induction
  - Maximum subarray
  - Matrix multiply, Strassen's method
  - **Master method of solving recurrences**
- Probabilistic Analysis (ch5)
  - Hiring problem and analysis
  - Randomised algorithms and PRNGs

Master method for recurrences

- If T(n) has the **form**: a T(n/b) + f(n), with a, b > 0
  - **Merge sort**: a = 2, b = 2, f(n) = Θ(n)
- Case 1: if $f(n) \in \Theta\left(n^{\log_b a}\right)$
  - Leaves/roots **balanced**: $T(n) = \Theta\left(n^{\log_b a} \log n\right)$
- Case 2: if $f(n) \in O\left(n^{\log_b a-\varepsilon}\right)$ for some z > 0
  - **Leaves** dominate: $T(n) = \Theta\left(n^{\log_b a}\right)$
- Case 3: if $f(n) \in \Omega\left(n^{\log_b a+\varepsilon}\right)$ for some z > 0, **and** if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some c < 1 and big n
  - **Roots** dominate: $T(n) = \Theta(f(n))$
  - Polynomials $f(n) = n^k$ satisfy the **regularity** condition

Master method: merge sort

- **Recurrence**: T(n) = 2T(n/2) + Θ(n):
  - a = 2, b = 2, f(n) = Θ(n)
- $f(n) = \Theta(n) = \Theta\left(n^{\log_2 2}\right)$
  - So leaves and roots are **balanced** (case 1)
- **Solution** is $T(n) = \Theta\left(n^{\log_2 2} \log n\right) = \Theta(n \log n)$

Master method: Strassen

- **Recurrence**: T(n) = 7T(n/2) + Θ(n^2)
  - a = 7, b = 2, f(n) = Θ(n^2)
- $f(n) = \Theta\left(n^2\right) = O\left(n^{\log_2 7-\varepsilon}\right)$
  - lg 7 ≃ 2.8, so, e.g., z = 0.4 works
  - So **leaves** dominate (case 2)
- **Solution** is $T(n) = \Theta\left(n^{\log_2 7}\right) \approx \Theta\left(n^{2.8}\right)$

Gaps in master method

- **Doesn't** cover all recurrences of form a T(n/b) + f(n)!
  - e.g., T(n) = 2T(n/2) + n log n
  - **Case 1**: $n \log n \notin \Theta\left(n^{\log_2 2}\right) = \Theta(n)$
  - **Case 2**: $n \log n \notin O\left(n^{1-\varepsilon}\right)$, for **any** z > 0
  - **Case 3**: $n \log n \notin \Omega\left(n^{1+\varepsilon}\right)$, for **any** z > 0
    - because $\log n \notin \Omega(n^{\varepsilon})$ ∀ z > 0
- Need to use **other** methods to solve
  - Some recurrences are just **intractable**

Polylog extension

- **Generalisation** of master method
- Applies for $f(n) \in \Theta\left(n^{\log_b(a)} \log^k(n)\right)$
  - (log to k power, not iterated log)
- **Solution**: $T(n) = \Theta\left(n^{\log_b(a)} \log^{k+1}(n)\right)$
  - **Regular** master method is special case, k = 0
- Previous **example**: T(n) = 2T(n/2) + n log n
  - **Solution**: $T(n) = \Theta\left(n \log^2 n\right)$

Outline for today

- Divide and conquer (ch4)
  - Merge sort, recursion tree
  - Proof by induction
  - Maximum subarray
  - Matrix multiply, Strassen's method
  - Master method of solving recurrences
- **Probabilistic Analysis** (ch5)
  - **Hiring problem and analysis**
  - **Randomised algorithms and PRNGs**

Probabilistic analysis

- Running time of **insertion sort** depended on input
  - Best-case vs worst-case vs **average**-case
- **Random variable** X: takes values within a domain
  - **Domain** Ω could be `[0,1]`, $\mathbb{R} = (-\infty, \infty)$, $\mathbb{R}^n$, `(A, A-, B+, ...)`, `{blue, red, black}`, etc.
- **Distribution** P(X): says which values are more likely
  - **Uniform**: all values equally likely
  - **Normal** (Gaussian) “bell curve” N(μ, σ)
- **Expected value** E(X): weighted average
  - $E(X) = \int_{X \in \Omega} P(X) = \sum_{X \in \Omega} P(X)$

Example: hiring problem

- **Input**: list of candidates with suitability $\{s_i\}_{i=1}^n$
  - cost per interview: c_i. cost per hire: $c_h > c_i$
- **Output**: list of hiring decisions $\{X_i\} \in \{0, 1\}^n$
  - **Constraint**: at any point, best candidate so far is hired
  - **Goal**: minimise total cost of interviews + hires
- Total **cost** is: $c_i n + c_h \sum_{i=1}^{n} X_i$
  - Interview cost is **fixed**, so focus on hiring cost
- **Worst** case: every new candidate is hired: $X_i = 1 \forall i$
  - (What kind of suitabilities $\{s_i\}$ would cause this?)

Analysis of hiring problem

- **Assume** order of candidates is random
  - each of n! possible **permutations** is equally likely
- For each candidate i, find probability of being **hired**:
  - Most **suitable** candidate seen so far
  - s_i needs to be **max** of $\{s_k\}_{k=1}^{i}$
  - if order is **random**, likelihood is 1/i
  - So $P(X_i) = \frac{1}{i}$
- Now we can derive the **expected hiring cost**

Expected hiring cost

- $E\left[c_h \sum_{i=1}^{n} X_i\right] = c_h \sum_{i=1}^{n} E[X_i]$ (by **linearity** of E)
- $= c_h \sum_{i=1}^{n} P(X_i)$ (since X_i is an **indicator**)
- $= c_h \sum_{i=1}^{n} \frac{1}{i}$ (random **order**, see prev slide)
- $= c_h (\ln n + O(1))$ (**harmonic series**)
- $\Rightarrow$ much better than **worst-case**: $c_h n$

Randomised algorithms

- Above analysis assumed input order was **random**
  - But we **can't** always assume that!
- So **inject** randomness into the problem:
  - **Shuffle** input before running algorithm
- Use a **pseudo-random** number generator (PRNG)
  - Typically, returns a **float** in range `[0,1)`
  - Sequence is reproducible by setting **seed**
- Or **hardware** RNG module (on motherboard, USB, etc.)
  - shot noise, Zener diode noise, beam splitters, etc.

Fisher-Yates shuffle

- Idea by **Fisher + Yates** (1938)
  - Implementation via swaps by **Durstenfeld** (1964)
- Randomly **permute** input A[] in-place, in O(n) time

```
def shuffle(A, n):
  for i in 1 to n:
    swap( A[ i ], A[ random( i, n ) ] )
```

- Use **PRNG** `random(a,b)`: int between a and b
- **Correctness** can be proved via loop invariant:
  - After i-th iteration, each possible permutation of length i is in the subarray A[1 .. i] with probability (n-i)!/n!

Outline for today

- **Divide and conquer** (ch4)
  - **Merge sort**, recursion tree
  - Proof by **induction**
  - Maximum **subarray**
  - Matrix multiply, **Strassen**'s method
  - **Master method** of solving recurrences
- **Probabilistic Analysis** (ch5)
  - **Hiring** problem and analysis
  - **Randomised** algorithms and PRNGs