9.1 RECAP: ELEMENTARY PROBABILITY THEORY

- Consider throwing two fair dice.
- The set of all outcomes (called sample space) given below

<table>
<thead>
<tr>
<th></th>
<th>D1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2</td>
<td>1</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td>(2,6)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>(3,1)</td>
<td>(3,2)</td>
<td>(3,3)</td>
<td>(3,4)</td>
<td>(3,5)</td>
<td>(3,6)</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>(4,1)</td>
<td>(4,2)</td>
<td>(4,3)</td>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>(5,1)</td>
<td>(5,2)</td>
<td>(5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>(6,1)</td>
<td>(6,2)</td>
<td>(6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
</tr>
</tbody>
</table>
Since dice are fair the probability that the die shows number $i$ is $1/6$, $i \in \{1,\ldots,6\}$

We wish to calculate sum of two dice is 5

There are 4 possibilities that sum of dice equals 5 which are: $(1,4),(2,3),(3,2),(4,1)$

So $P(\text{sum of two dice is 5}) = 4/36 = 1/9$
9.1.1 SAMPLE SPACES & PROBABILITY DISTRIBUTIONS

• Sample space (denoted by $S$) $\Rightarrow$ set of all outcomes in the experiment

• Event $\Rightarrow$ A subset of outcomes from $S$

• A probability distribution arises by $P(E) \Rightarrow$ each event $E$ in $S$

$$P(E) = \frac{|E|}{|S|} \Rightarrow P(E) \text{ of an event } E, \text{ if } S \text{ is finite AND}$$

Each event equally likely to occur

• Two events are *mutually independent* if $E \cap F = \emptyset$
A probability distribution $P$ on the sample space $S$ is a mapping from the events in $S$ to the real numbers satisfying the following

Axiom 1: $0 \leq P(E) \leq 1$

Axiom 2: $P(S) = 1$

Axiom 3: $P(\bigcup_{i=1}^{\infty} E_i) = \sum_{i=1}^{\infty} P(E_i)$
9.1.2 CONDITIONAL PROBABILITY

• Plays big role in finding average behaviour of algorithms

Example: Find the probability of that the sum of 2 dice is at most 5 given first die is 1

Solution: Denote $E \Rightarrow$ sum of 2 dice is at most 5
$F \Rightarrow$ first die is 1

$P(E|F) \Rightarrow$ cond. probability of $E$ given $F$

$P(E) = 10/36 = 5/18$ and $P(F) = 6/36 = 1/6$

$E \cap F = \{(1,1), (1,2), (1,3), (1,4)\}$ So,

$P(E|F) = \frac{P(E \cap F)}{P(F)} = \frac{4/6}{2/3}$
9.1.2 CONDITIONAL PROBABILITY

- General definition of conditional probability:
  \[ P(E|F) = \frac{P(E \cap F)}{P(F)} \]
- Events E and F are independent if
  \[ P(E|F) = P(E) \quad \text{or} \quad P(E \cap F) = P(E)P(F) \]
- By using mutually exclusiveness and axioms of probability
  Baye’s formula can be derived which is
  \[ P(E) = P(F) \cdot P(E|F) + (1 - P(F)) \cdot P(E|F^c) \]
9.1.3 RANDOM VARIABLES AND EXPECTATION

• A random variable $X$ on sample space $S$ is a mapping from $S$ to the set $\mathbb{R}$ of real numbers

$P(X=x) \Rightarrow$ prob. of occurrence of event $E = \{s \in S \mid X(s) = x\}$

$F(x) = P(X=x) \Leftarrow$ probability distribution function (PDF)

determines distribution of the random variable $X$

• Binomial Random variable $\Rightarrow$ number of successes associated with given outcome

$P(X=i) = \binom{n}{i} (1-p)^{n-i} p^i$
9.1.3 RANDOM VARIABLES AND EXPECTATION

• Geometric Random variable $\Rightarrow$ first success after $i$ trials
  
  $P(X=i) = (1-p)^{i-1} p$

• N-th Moment
  
  $E[X^N] = \sum_x x^N P(X=i)$

  N=0, checks whether the function is pdf or not,
  N=1 gives the mean or expectation of $X$,
  N=2, will be used for variance

  • Properties:
    
    $E[cX]=c \ E[X]$ where $c$ is a constant

    $E[X]= E[X_1]+ E[X_2]+\ldots\ldots\ldots E[X_n]$ if $X= X_1+X_2+\ldots+X_n$

• It is also useful to know how much $X$ deviates from $E[X]$
9.1.3 RANDOM VARIABLES AND EXPECTATION

- The deviation of $X$ from $E[X]$ is formally defined in terms of variance: $V[X] = E[(X - E[X])^2]$

- This can be simplified as: $V[X] = E[X^2] - (E[X])^2$

- Conditional expectation $E[X|F]$ to be:
  $$E[X|F] = \sum_{s \in F} X(s) P(s|F)(s) P(s|F)$$ or equivalently
  $$E[X|F] = \sum_x x P(X = x|F)$$

**Example:** Consider rolling of two dice and let $X$ sum of two dice i.e. $d_1 + d_2$ Suppose $F$ is the event that $d_1 = 2$. Since $d_1$ and $d_2$ are independent:

$$P((2,d2)|F) = 1/6$$
9.1.3 RANDOM VARIABLES AND EXPECTATION

Example ctd. Let’s find $E[X|F]$

$$(s)P(s|F)(s) = 1/6 \sum_{d2 \in \{1..6\}} (2+d2)$$

$$=(1/6)(3+4+5+6+7+8) = 5.5$$

• Let $x$ be a random variable on sample space $S$ and we partition $S$ into disjoint subsets $F_i$, $i=1..m$. Then we have:

$$E[X]=\sum_{i=1}^{m} E[X|F_i]P(F_i)$$

• We also have two following propositions

$$E[X]=\sum_{Y=y} E[X|Y=y]P(Y=y)$$

$$E[X|Y=y]=\sum_{x} xP(X=x|Y=y)$$
9.2 AVERAGE COMPLEXITY REVISITED

- When analyzing the complexity of an algorithm, the critical issue is often average complexity.

**Average complexity** ⇒

$I_n$ → set of all inputs of size $n$ to a given algorithm.

$\tau(I)$ → # of basic operations performed by alg. on input $I$. $I \in I_n$

Given $I_n$, the average complexity:

$$A(n) = E[\tau]$$
9.2.1 TECHNIQUES FOR COMPUTING AVERAGE COMPLEXITY

• Depending on the characteristics of the algorithm one or some combinations of formulas will be most applicable for $A(n)$

**FORMULA 1**

$$A(n) = E[\tau] = \sum_{I \in \mathcal{J}_n} \tau(I)P(I)$$

Rarely used since it is too cumbersome to examine each element in $\mathcal{J}_n$.

**FORMULA 2**

$$A(n) = E[\tau] = \sum_{i=1}^{W(n)} iP(\tau = i)$$

$P(\tau = i) \Rightarrow$ prob that alg performs exactly $i$ basic operations.
9.2.1 Techniques for Computing Average Complexity

**Formula 3**

\[ A(n) = E[\tau] = \sum_{i=1}^{W(n)} P(\tau \geq i) \]

\[ P(\tau \geq i) \Rightarrow \text{prob that alg performs at least } i \text{ basic operations} \]

**Formula 4**

\[ A(n) = E[\tau] = \sum_{i=1}^{k} E[\tau_i] \]

**Formula 5**

\[ A(n) = E[\tau] = \sum_{y} E[\tau|Y=y] \cdot P(Y=y) \]

• When determining which formulation to use, use following techniques:

  1. Partitioning the algorithm
  2. Partitioning input space
  3. Recursion
9.3 AVERAGE COMPLEXITY OF LINEAR SEARCH

- Let the list consists of distinct elements and the search element $X$ occurs with a probability $p$
- $p_i \Rightarrow X$ is in the $i$th element in $L[i]$
- $P_i = P(X = L[i] | X \text{ is in the list})$
  $P(X \text{ is in the list}) = (1/n)p$
- Use formula 2 to obtain:
  $$A(n) = 1(p/n) + 2(p/n) + \ldots + (n-1)(p/n) + n((p/n)+1-p)$$
  $$= (1-(p/2))n + p/2$$
  n comparisons when $X$ is in $L[n]$ or $X$ is not in the list
9.3.1 AVERAGE COMPLEXITY OF LINEAR SEARCH WITH REPEATED ELEMENTS

• Determine $A(n,m)$ where $m$ is # of distinct elements

• $L[i]$ has equal probability of $1/m$ in $S$

• The prob. that $X$ does not occur in position $i \Rightarrow (m-1)/m$

• The prob. that $X$ is not in the list $\Rightarrow ((m-1)/m)^n$

• The prob. that $X$ is in the list $\Rightarrow 1 - ((m-1)/m)^n$

• $p_i \Rightarrow$ first occurrence of search element $X$ in position $i$
9.3.1 AVERAGE COMPLEXITY OF LINEAR SEARCH WITH REPEATED ELEMENTS

\[ p_i = \begin{cases} 
((m-1)/m)^{i-1}(1/m) & \text{if } 1 \leq i \leq n-1 \\
((m-1)/m)^n & \text{if } i = n 
\end{cases} \]

- Substitute formula 2 to obtain

\[ A(n,m) = \sum_{i=1}^{\text{w}(n)} w_i p_i = \sum_{i=1}^{n-1} i \cdot \left( (m-1)/m \right)^{i-1}(1/m) + ((m-1)/m)^n n \]

- By simplifying we obtain

\[ m(1 - ((m-1)/m)^n + (m-1)/m)^{n-1} \]
9.3.1 AVERAGE COMPLEXITY OF INSERTION SORT

• Inputs of insertion sort is permutations of \{1,2...,n\}
• We have to partition the algorithm into n-1 stages
• The i^{th} stage consists of inserting the (i+1)^{th} element L[i+1] into its proper place in the list L[1],..,L[i]

\[ \text{A(n)} = E[\tau] = E[\tau_1] + E[\tau_2] + .. + E[\tau_{n-1}] \text{ where } \tau_i \Rightarrow \text{# of comparasions in } i^{th} \text{ stage} \]

• Calculate \( E[\tau_i] \) with 
  \[ E[\tau_i] = \sum_{j=1}^{i} jP(\tau=j) \]

1. \( P(\tau=j) = \frac{1}{i+1}, \quad j=1,\ldots,i-1 \)
2. \( P(\tau=i) = \frac{2}{i+1}, \quad j=1,\ldots,n-1 \)
9.3.1 AVERAGE COMPLEXITY OF INSERTION SORT

• Substitude 1 and 2 into previous formula and simplify to get:
  \[ E[\tau_i] = \left( \sum_{j=1}^{i} \frac{j}{i+1} \right) + \frac{i}{i+1} = \left( \frac{i}{2} \right) + 1 - \left( \frac{1}{i+1} \right) \]

• Substitude this to our first formula to find \( A(n) \):
  \[ \sum_{i=1}^{n-1} \left( \left( \frac{i}{2} \right) + 1 - \left( \frac{1}{i+1} \right) \right) \]
  \[ A(n) = (n-1)n/4 + n - H(n) \]
  , where \( H(n) \) is harmonic series
  \[ 1 + 1/2 + 1/3 + \ldots + 1/n \approx \ln n \]
Chapter 9

9.5 Average Complexity of QuickSort
9.6 Average Complexity of MaxMin2
9.7 Average Complexity of BinarySearch and Bin SrchTreeSearch
Average Complexity of QuickSort

Assumptions:

- The input lists $L[1:n]$ to \textit{QuickSort} are all permutations of $1, 2, \ldots, n$, with each permutation being equally likely.

Remember the algorithm of \textit{QuickSort}:

- We partition QuickSort into two stages, where the first stage is the call to \textit{RearrangeAndPlace} and
- the second stage is the \textbf{two recursive calls} with input lists consisting of the sublists on either side of the proper placement of pivot element $L[1]$. 
Average Complexity of QuickSort (2)

- **RearrangeAndPlace (named as T1)**
  Rearranges the list with respect to previously chosen pivot element.
  So \( (n+1) \) comparisons are performed.

- **Recursive Calls (named as T2)**
  Sort sublists \( L[1: i-1] \) and \( L[i+1: n] \) with \( n \) different choices of \( i \), \( (i\text{th element as pivot}) \)
Average Complexity of QuickSort (3)

Formulation

\[ A(n) = E[T] \]
\[ = E[T_1] + E[T_2] \]
\[ = (n+1) + \frac{1}{n} \sum (A(i-1) + A(n-i)) \]
\[ = (n+1) + \frac{2}{n} (A(0) + A(1) + \ldots + A(n-1)) \]
\[ A(0) = A(1) = 0. \]
Average Complexity of QuickSort (4)

\[ nA(n) = n(n+1) + 2(A(0)+A(1)+...+A(n-1)) \]

Substituting \( n-1 \) for \( n \) in the previous formula

\[ (n-1)A(n-1) = n(n-1) + 2(A(0)+A(1)+...+A(n-2)) \]

Subtract

\[ nA(n)-(n-1)A(n-1) = 2n + 2A(n-1) \]

\[ A(n)/(n+1) = A(n-1)/n + 2/(n+1) \]

Let \( t(n) = A(n)/(n+1) \)

\[ t(n) = t(n-1) + 2/(n+1), \quad t(1)=0 \]

Solve the above recurrence relation...
Average Complexity of QuickSort (5)

\[ t(n) = 2\left(\frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n+1}\right) = 2H(n+1) - 3 \sim 2\ln \]

Remember that \( t(n) = A(n)/(n+1) \)
\[ A(n) \sim 2n\ln \]

In particular, *QuickSort* exhibits \( O(n\log n) \) average behavior, which is the order optimal for a comparison based algorithm.
Average Complexity of MaxMin2

Pseudo code

procedure MaxMin2 (L[1:n], MaxValue, MinValue)
MaxValue:=L[1]
MinValue:=L[1]
for i:=2 to n do
    if L[i]>MaxValue then
        MaxValue:=L[i]
    else
        if L[i]<MinValue then
            MinValue:=L[i]
        endif
    endif
endfor
end MaxMin2
Average Complexity of MaxMin2(2)

Assumptions:
- The inputs permutations of 1,2,...,n, and that each permutation is equally likely.
- We already know that
  \[ B(n) = n-1 \quad W(n) = 2(n-1) \]

(n-1) comparisons involving MaxValue are performed for any input permutation.
An additional comparison involving MinValue is performed for each iteration of the loop in which MaxValue is not updated.
Average Complexity of MaxMin2(3)

- **D**: random variable that denotes the number of times that MaxValue is updated.
- **T** = n-1 + (n-1-D) = 2n-2-D
- **A(n) = E[T] = 2n-2-E[D]**

We compute the expected number of updates **E[D]** by partitioning the input space by utilizing the r.v M.

**E[D]=∑ E[D|M=i] P(M=i)** where P(M=i) = 1/n i=1,2,..,n

- **E[D|M=i] = α(n) = (1/n)(α(n-1) +1) + ((n-1)/n)α(n-1)**
- **α(n) = ½+1/3 +...+1/n = H(n)-1**
- **A(n) = 2n-2 - α(n) = 2n-H(n)-1**

**A(n) = 2n - ln n - 1 ~ W(n) = O(n)**
Average Complexity of BinarySearch

- Remember **BinarySearch** given in Ch.3, we choose the 3-brach comparison of the **do case** statement as the basic operation:

  ```
  case
  \hspace{1cm} X = L[mid] : Found := true.
  \hspace{1cm} X < L[mid] : high := mid-1
  \hspace{1cm} otherwise : low := mid+1
  endcase
  ```
**Average Complexity of BinarySearch (2)**

**Assumptions:**
- **p:** the probability that the search element X is on the list.
- Given that X is *on the list* \( L[1:n] \); we assume that it is equally likely to occur in any of n positions.
- Given that X is *not on the list*; we assume it is equally likely to occur in any of the \( (n+1) \) intervals.

\( X < L[1], \ L[1] < X < L[2], \ldots L[n-1] < X < L[n], \ X > L[n] \)
Average Complexity of BinarySearch (3)

- The probability that X occurs on the list and is equal to any given element \( L[i] \) : \( \frac{p}{n} \)

- The probability that X does not occur in any of the \( n+1 \) intervals : \( \frac{1-p}{n+1} \)
**Average Complexity of Binary Search (4)**

**Reminder (properties of Binary Search Tree)**

- **IPL: internal path length**, the sum of the lengths of the paths from root to the internal nodes as the internal nodes vary over the entire tree.
- **LPL: leaf path length**, defined similarly.

Note: length of a path from the root to a node at level $i$:

$$i$$

(from Ch.7, proposition 7.3.6)

$$\text{IPL}(T) = \text{LPL}(T) - 2I,$$

where $I$ is the number of internal nodes.
Average Complexity of Binary Search (5)

Thus $IPL(T) = LPL(T) - 2n$

So, $A(n) = \frac{p}{n}(IPL(T)+n) + \frac{1-p}{n+1}LPL(T)$

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

$= \left(\frac{p}{n} + \frac{1-p}{n+1}\right)LPL(T) - p$

(from Ch.7, proposition 7.3.7)

$LPL(T) = L \text{ floor } (\log L) + 2(L^\text{power}(2,\log L))$, if $T$ is a 2-tree and is full at the second-deepest level.

$LPL(T) >= L \text{ ceiling } (\log L) = \text{ceiling } ( (n+1)\log(n+1) )$

$A(n) >= \left(\frac{p}{n} + \frac{1-p}{n+1}\right)( (n+1)\log(n+1) ) - p$
Average Complexity of BinarySearch & BinSrchTreeSearch

- The lower bound estimate for $A(n)$
  $A(n) \sim W(n) = \log(n+1)$

- For BinSrchTreeSearch,
  $A(n) = LPL(T) \left( \frac{1+ (p/n)}{(n+1)} \right) - p$

......

$A(n) \sim \Omega(\log n)$, apply thm 7.3.7
Chapter 7 old version, Chapter 3 in new version

Introduction, Correctness Proofs, and Recurrence Relation
Table of Contents

7.1 Mathematical Induction
   - 7.1.1 Principle of Mathematical Induction
   - 7.1.2 Variations of the Principle of Mathematical Induction

7.2 Correctness Proofs
   - 7.2.1 Loop Invariants
   - 7.2.2 Induction on Input Size
   - 7.2.3 Correctness Proofs for Parallel Algorithms: The 0/1 Sorting Lemma

7.3 Mathematical Properties of Binary Trees
   - 7.3.1 Lower Bounds for the Depth of Binary Trees
   - 7.3.2 Internal and Leaf Path Lengths of Binary Trees

7.4 Solving Recurrence Relations for Complexity
   - 7.4.1 Solving Linear Recurrence Relations Relating the \( n \)th Term to \( (n-1) \)st Term
   - 7.4.2 Solving Linear Recurrence Relations Relating the \( n \)th Term to \( (n/b) \)th Term
   - 7.4.3 Interpolating Asymptotic Behavior
7.1 Mathematical Induction

Often we would like to establish the validity of a proposition or formula concerning the set of positive integers such as;

\[ 1^2 + 2^2 + \ldots + n^2 = \frac{n (n + 1) (2n + 1)}{6} \]

This formula can be easily verified for any particular small \( n \) by direct computing. The question is how do we verify that the formula is true for all positive integers \( n \)?
7.1.1 Principle of Mathematical Induction

- **Basis step:** P(1) is true
- **Induction step:** if P(k) is true for any given k, then P(k+1) must also be true

Now let's prove the previous equation:

**Basis step:** \(1^2 = \frac{1 \cdot (1 + 1) \cdot (2 + 1)}{6}\) ... is true

**Induction step:** \(1^2 + 2^2 + \ldots + k^2 + (k + 1)^2 = (1^2 + 2^2 + \ldots + k^2) + (k + 1)^2\) = \(\frac{k \cdot (k + 1) \cdot (2k + 1) + (k + 1)^2}{6}\) = \(\frac{(k + 1) \cdot [k \cdot (2k + 1) + 6(k + 1)]}{6}\) = \(\frac{(k + 1) \cdot (k + 2) \cdot (2k + 3)}{6}\) ... and therefore P(k+1) is true
3 variations of mathematical induction are frequently encountered in the analysis of algorithms

1. The sequence of propositions starts with an index different from 1, such as 0. Then the basis step starts with this initial index. The induction step remains the same, and the two steps together establish the truth of the proposition.

2. The propositions are only finite in number, P(1), …, P(l). Then the induction step is modified to require that k < l. The conclusion then drawn is that P(1), …, P(l) are all true if the basis and induction steps are valid.

3. Induction Step (strong form) : For any positive integer k, if p(j) is true for all positive integers j <= k, then P(k+1) must also be true.
A complete analysis of an algorithm should not only describe its complexity, but should also include a verification of its correctness.

A technique for verifying correctness is to identify certain loop invariants associated with loops in the pseudocode for an algorithm.

A loop invariant is a statement about the value of a variable or condition after each iteration of the loop.
7.2.1 Loop Invariants

To illustrate the idea of loop invariants, consider the algorithm EgyptianPowers given below;

Function EgyptianPowers(x,n)
Input : x (a real number), n (a power of two, n = 2m)
Output : xn
j := 1
Product := x
while j<n do
    Product := Product * Product
    j := j + j
end while
Return (Product)
End EgyptianPowers

Basis step : After k=0 passes, j has the value 1 = 2^0 and Product has the value x=x^2. Thus, the two loop invariants have the stated values.

Induction step : Assume that after k passes, j has the value 2^k, and Product has the value x^j, for k<m. Then j = 2^k < 2^m = n, and another pass through the while loop is performed. During each pass of the while loop in Egyptian Powers, j is doubled, and Product is squared. Thus after k+1 passes, j has the value 2^k+2^k=2^{k+1} and Product has the value x^{2^k}x^{2^k} = x^{2^{k+1}} , completing the induction step.
7.2.2 Induction on Input Size

To illustrate a correctness proof that uses induction on the input size, we consider a variation of the algorithm BinarySearch.

**Function** BinarySearch2(L[low,high], X) **recursive**

**Input**: L[low,high], X (search item)

**Output**: Returns the index of an occurrence of X in the list, or 0 if X is not in the list.

```plaintext
if low = high then
    if X = L[low] then
        return(low)
    else
        return(0)
    endif
else
    mid := (low+high+1) / 2
    if X < L[mid] then
        return(BinarySearch2(L[low:mid-1], X))
    else
        return(BinarySearch2(L[mid:high], X))
    endif
endif
End BinarySearch2
```
7.2.2 Induction on Input Size

Correctness proof of BinarySearch2;

- **Basis step**: If \( k = 1 \), then \( \text{low} = \text{high} \), so that the sublist \( L[\text{low} : \text{high}] \) consists of the single element \( L[\text{low}] \), and the algorithm correctly compares \( X \) to \( L[\text{low}] \).

- **Induction step**: Assume BinarySearch2 works correctly on all sublists \( L[\text{low} : \text{high}] \) of size \( k = \text{high} - \text{low} + 1 \), for \( 1 \leq k < n \).
  
  Now consider a sublist \( L[\text{low} : \text{high}] \) of size \( k+1 \).
  
  Since \( k+1 > 1 \), \( \text{low} \neq \text{high} \), so that \( \text{mid} \) is the assigned the value \( (\text{low} + \text{high} + 1) / 2 \) and \( X \) is compared to \( L[\text{mid}] \). If \( X < L[\text{mid}] \), then BinarySearch2 invokes itself recursively. We consider the case where \( X < L[\text{mid}] \). Since \( L \) is ordered, this means that \( X \) does not occur in the sublist \( L[\text{mid} : \text{high}] \). Thus, if \( X \) occurs at all in \( L[\text{low} : \text{high}] \), then it must occur in the sublist \( L[\text{low} : \text{mid}-1] \). Since \( \text{mid} - 1 < \text{high} \), the size of the latter sublist is at most \( k \).

  Thus, BinarySearch2 works correctly on \( L[\text{low} : \text{high}] \).
7.2.3 Correctness Proofs for Parallel Algorithms: The 0/1 Sorting Lemma

Many simply described parallel sorting algorithms, such as OddEvenMergeSort1DMesh are surprisingly difficult to prove for a general input list. However, the powerful 0/1 Sorting Lemma allows us to restrict attention to 0/1 lists when establishing correctness.

0/1 Sorting Lemma: Suppose an oblivious comparison-exchange sorting algorithm works correctly on all lists of size $n$ consisting of only 0s and 1s. Then it works correctly on all lists of size $n$. 
The analysis of the complexity of many problems and algorithms discussed in this text depends on various mathematical properties of the binary trees. In this section we establish a number of these properties relating to depth, internal path length and leaf path length.
7.3.1 Lower Bounds for the Depth of Binary Trees

Given any binary tree $T$, we will use the notation:

- $N = N(T)$ → number of nodes
- $I = I(T)$ → number of internal nodes
- $L = L(T)$ → number of leaf nodes
- $D = D(T)$ → depth of tree
7.3.1 Lower Bounds for the Depth of Binary Trees - Propositions

- **Proposition 1**: Suppose $T$ is any binary tree having $N$ nodes. Then $T$ has depth at least:

\[ \lceil \log_2 N \rceil \]

- **Proposition 2**: Suppose $T$ is any binary tree. Then the number of leaf nodes is one greater than the number of internal nodes of $T$; that is,

\[ I(T) = L(T) - 1 \quad \text{(equivalently we have } N(T) = 2L(T) - 1) \]

- **Proposition 3**: Suppose $T$ is any binary tree. Then the depth of $T$ satisfies,

\[ D(T) \geq \lceil \log_2 L(T) \rceil \]

- **Proposition 4**: Suppose $T$ is any binary tree. Then $T$ is full at the second-deepest level if, and only if, all the leaf nodes are contained in two levels ($D-1$ and $D$).

- **Proposition 5**: If a binary tree is full at the second-deepest level, then the depth $D(T)$ and the number of leaf nodes $L(T)$ are related by

\[ D(T) = \lceil \log_2 L(T) \rceil \]
7.3.2 Internal and Leaf Path Lengths of Binary Trees

The *internal path length* $\text{IPL}(T)$ of a binary tree $T$ is defined as the sum of the lengths of the paths from the root to the internal nodes as the internal nodes vary over the entire tree.

The *leaf path length* $\text{LPL}(T)$ of a binary tree $T$ is defined as the sum of the lengths of the paths from the root to the leaf nodes.
7.3.2 Internal and Leaf Path Lengths of Binary Trees - Propositions

- **Proposition 6:** Given any 2-tree $T$ having $I$ internal nodes.
  \[ \text{IPL}(T) = \text{LPL}(T) - 2I \]

- **Proposition 7:** Given any 2-tree $T$ with $L$ leaf nodes,
  \[ \text{LPL}(T) \geq L \lfloor \log_2 L \rfloor + 2(L - 2 \lceil \log_2 L \rceil) \]

- **Corollary 1:** If $T$ is any binary tree having $L$ leaf nodes, then
  \[ \text{LPL}(T) \geq \lceil L \log_2 L \rceil \]
  Further, if $T$ is a full binary tree, then inequality is an equality.
A linear recurrence relation is one of the form:

\[ t(n) = c_1 t(n-1) + c_2 t(n-2) + \ldots + c_k t(n-k) + f(n), \]

initial condition; \( t(0) = d_0, \ldots, t(k-1) = d_{k-1} \),
for constants \( c_1, \ldots, c_k, d_0, \ldots, d_{k-1} \) and
some fixed function \( f(n) \)
7.4.1 Solving Linear Recurrence Relations
Relating the \( n \)th Term to \((n-1)\)st Term

**General Recurrence Relation**;
\[
t(n) = at(n-1) + f(n)
\]
\[
t(n) = a(at(n-2) + f(n-1)) + f(n) = a^2t(n-2) + af(n-1) + f(n)
\]
\[
t(n) = a^2(at(n-3) + f(n-2)) + af(n-1) + f(n)
\]
\[
t(n) = a^3t(n-3) + a^2f(n-2) + af(n-1) + f(n)
\]
\[
\vdots
\]
\[
t(n) = a^{n-1}b + \sum_{i=2}^{n} a^{n-i} f(i)
\]
General Recurrence Relation;
\[ t(n) = at(n/b) + f(n) \]
\[ t(n) = a(at(n/b^2) + f(n/b)) + f(n) = a^2t(n/b^2) + af(n/b) + f(n) \]
\[ t(n) = a^k c_1 + a^{k-1}f(n/b^{k-1}) + a^{k-2}f(n/b^{k-2}) + \ldots + af(n/b) + f(n) \]
\[ t(n) = a^k c_1 + \sum_{i=0}^{k-1} a^i f(n/b^i), \text{ where } k = \log_b n \]

Special case \( f(n) = cn; \)
\[ t(n) = \begin{cases} 
  c_1 n + cn \log_b n & \text{if } a=b \\
  c_1 n^{\log_b a} + c (n^{\log_b a} - n) / (a/b) - 1 & \text{if } a \neq b 
\end{cases} \]
7.4.3 Interpolating Asymptotic Behavior

Proposition 1: If \( g(n) \in \mathcal{F} \) has the property that \( g(cn) \in \Theta(g(n)) \), then \( f(cn) \in \Theta(g(n)) = \Theta(f(n)) \) for all \( f(n) \in \Theta(g(n)) \).

Proposition 2: Suppose \( g(n) \in \mathcal{F} \) is eventually nondecreasing and \( \Theta \)-invariant under scaling. Further, suppose \( f(n) \in \mathcal{F} \) is eventually nondecreasing and \( f(b^n) \in X(g(b^n)), n \in \mathbb{N} \), where \( X \) is one of the classes \( \Theta, \Omega, O \), and \( b > 1 \). Then \( f(n) \in X(g(n)) \).

Proposition 3: If \( t(n) \) is eventually nondecreasing and satisfies the recurrence relation \( t(n) = at(n/b) + c \), with initial condition \( t(1) = c_1 \), where \( a, b, c, c_1 \) are positive constants, \( b > 1 \), then
\[
\Theta(n^{\log_b a}), \quad a \neq 1 \\
\Theta(\log n), \quad a = 1
\]

Proposition 4: If \( t(n) \) is eventually nondecreasing and satisfies the recurrence relation \( t(n) = at(n/b) + cn \), with initial condition \( t(1) = c_1 \), where \( a, b, c, c_1 \) are positive constants, \( b > 1 \), then
\[
\begin{cases}
\Theta(n), & a < b \\
\Theta(n \log n), & a = b \\
\Theta(n^{\log_b a}), & a > b
\end{cases}
\]
Consider the recurrence relation in the form of

\[ C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i} \quad \text{for } n \geq 0; \]

\[ C_n = ??? \]
7.5 Generating Functions

7.5.1 The Generating Function Paradigm for Solving Recurrence Relations

7.5.2 Obtaining a Formula for the Number of Binary Trees on N Nodes

7.6 Polynomial Interpolation

7.6.1 Lagrange Interpolation
7.6.2 Newtonian Interpolation
Generating Functions

Multiple reference recurrence relations like \( \text{fib}(n)=\text{fib}(n-1)+\text{fib}(n-2), \) \( \text{fib}(0)=0, \)
\( \text{fib}(1)=1, \) determining fibonacci numbers 0,1,1,2,3,5,8,13,... or \( t(n)=2t(n-1)+1, \)
t(0)=0 for Towers of Hanoi problem, can be solved by using Generating functions.

For a sequence of numbers \( a_0, a_1, a_2, \ldots, a_n, \ldots \) the Generating Function is the formal power series:

\[
g(x) = \sum_{n=0}^{\infty} a_n x^n = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n + \ldots
\]
Generating Functions

\[ g(x) = \sum_{n=0}^{\infty} a_n x^n = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n + \ldots \]

e.g. Generating function for Fibonacci sequence is the power series

\[ x + x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 + 13x^7 + 21x^8 + \ldots \]

The generating function for a sequence \((a_n)\) is simply used to “encode” the sequence in a form that is amenable to algebraic manipulation. For Fibonacci sequence we will get the explicit formula:

\[ \text{fib}(n) = \left( \frac{1}{\sqrt{5}} \right) \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right] \]
How to Solve a Given Recurrence Relation

Suppose we have \( t(n) = c_1 t(n-1) + c_2 t(n-2) \), where \( t(0) = b_0 \), \( t(1) = b_1 \).

i) Form \( g(x) = \sum_{n=0}^{\infty} t(n)x^n \)

ii) Convert \( g(x) \) into a closed form formula by doing the following procedure:

\[
\begin{align*}
g(x) &= t(0) + t(1)x + t(2)x^2 + t(3)x^3 + \ldots \ldots \\
-c_1xg(x) &= -c_1t(0)x - c_1t(1)x^2 - c_1t(2)x^3 + \ldots \ldots \\
-c_2x^2g(x) &= -c_2t(0)x^2 - c_2t(1)x^3 + \ldots \ldots \\
\end{align*}
\]

\[
g(x)[1-c_1x+c_2x^2] = t(0) + x[t(1)-c_1t(0)] + x^2[t(2)-c_1t(1)-c_2t(0)] + \ldots \ldots 
\]

By definition \( t(2) = c_1 t(1) + c_2 t(0) \) and so on.

\[
g(x) = \frac{(b_0 + x[b_1 - c_1 b_0])}{(1-c_1x+c_2x^2)}
\]

Then solve by finding roots (by method of partial fractions).

iii) Expand \( g(x) \) into a power series with explicit coefficients:

\[
g(x) = \sum_{n=0}^{\infty} d(n)x^n \quad \text{Here } d(n) \text{ corresponds to } t(n) \text{ in step (i)}
\]
Example: Solve Towers of Hanoi

\[ t(n) = 2t(n-1) + 1, \quad t(0) = 0 \]

\[ g(x) = \sum_{n=0}^{\infty} t(n)x^n = 0 + 3x^2 + 7x^3 + 15x^4 + \ldots + a_nx^n + \ldots \]

\[ -2xg(x) = -2x^2 - 6x^3 - 14x^4 + \ldots \]

\[ g(x)[1-2x] = x[1 + x + x^2 + x^3 + x^4 + \ldots + x^{n-1} + \ldots] \]

Since \(1 + x + x^2 + x^3 + \ldots + x^{n-1} + \ldots = 1/(1-x)\)

\[ g(x) = \frac{X}{(1-2x)(1-x)} = \frac{1}{1-2x} - \frac{1}{1-x} \]

\[ g(x) = \sum_{n=0}^{\infty} [(2x)^n - x^n] = \sum_{n=0}^{\infty} (2^n - 1)x^n \]

And \(2^n - 1\) corresponds to \(t(n)\)

For \(t(n) = 2t(n-1) + 1, \quad t(0) = 0\)

we have \(t(n) = 2^n - 1, \quad n \geq 0\)
Formula for General Linear Recurrences

For any linear recurrence relation of type
\[ t(n) = c_1 t(n-1) + c_2 t(n-2) + \ldots + c_k t(n-k) \]
initial conditions \( t(0) = b_0, \ldots, t(k) = b_k \)

We can derive a formula using the method described above.

\[ g(x) = \frac{b_0 + (b_1 - c_1 b_0)x + \ldots + (b_{k-1} - c_{k-1} b_{k-1} - \ldots - c_k b_0)x^{k-1}}{1 - c_1 x - \ldots - c_k x^k} \]
Obtaining a Formula
for the Number of Binary Trees on N Nodes

The number \( b_n \) of binary trees on \( n \) nodes equals the \( n^{th} \) Catalan number:

\[
b_n = \frac{1}{(n+1)} \binom{2n}{n}
\]

\( b_n \) is given by the following quadratic recurrence relation:

\[
b_n = \sum_{i=0}^{n-1} b_i b_{n-1-i} = b_0 b_{n-1} + b_1 b_{n-2} + b_2 b_{n-3} + \ldots + b_{n-1} b_0
\]

Let \( g(x) \) be the generating function for the number \( b_n \) of binary trees on \( n \) nodes, that is:

\[
g(x) = \sum_{n=0}^{\infty} b_n x^n = b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n + \ldots
\]

\[
xg(x) = \sum_{n=0}^{\infty} b_n x^n = b_0 x + b_1 x^2 + \ldots + b_{n-1} x^n + \ldots
\]

\[
g(x)(xg(x)) = (g(x))^2 x = \sum_{n=1}^{\infty} \left( \sum_{i=0}^{n-1} b_i b_{n-1-i} \right) x^n =
\]

\[
= (b_0 b_0) x + (b_0 b_1 + b_1 b_0) x^2 + \ldots + (b_0 b_{n-1} + \ldots + b_{n-1} b_0) x^n + \ldots
\]

\[
(g(x))^2 x = b_1 x + b_2 x^2 + b_3 x^3 + \ldots + b_n x^n + \ldots = g(x) - 1
\]

Hence \((g(x))^2 x - g(x) - 1 = 0\)
Using the quadratic formula we have: 
\[ g(x) = \frac{1 \pm \sqrt{1 - 4x}}{2x} \]

Since \( b_0 = 1 \), it follows that: 
\[ g(x) = 1 - \frac{\sqrt{1 - 4x}}{2x} \]

Using the Binomial theorem \((1+x)^n = \sum_{i=0}^{\infty} \binom{n}{i} x^i\) we obtain

\[ g(x) = 1 - \frac{\sqrt{1 - 4x}}{2x} = -\sum_{n=1}^{\infty} \frac{1}{2} \binom{1/2}{n} \left(\frac{-4x}{2x}\right)^n \]

\[ = \sum_{n=0}^{\infty} \frac{1}{2} \binom{1/2}{n+1} (-2)^n (2)^{n+1} x^n \]

\[ = \sum \left[ \frac{(-1/2)(-3/2)(-5/2) \ldots (-2n-1/2)(-2)^{n+1} 2^n}{(n+1)!} \right] x^n \]

\[ = \sum \left[ \frac{(1)(3)(5)\ldots(2n-1)2^n}{(n+1)!} \right] x^n \]

\[ b_n = \frac{(1)(3)(5)\ldots(2n-1)2^n}{(n+1)!} \]

and \( (1)(3)(5)\ldots(2n-1)2^n = (n+1)(n+2)\ldots(2n-1)(2n) \) so,

\[ b_n = \frac{(n+1)(n+2)\ldots(2n-1)(2n)}{(n+1)!} = \frac{1}{(n+1)} \binom{2n}{n} \]
Polynomial Interpolation

We are asking for a unique (interpolating) polynomial of degree at most \( n-1 \) whose graph passes through \( n \) points \((x_1,y_1), (x_2,y_2), \ldots, (x_n,y_n)\),

e.g. For points \((-1,1),(0,0),(1,0),(3,2)\), we need to find a polynomial of degree at most 3 that interpolates these points.

Simply we need to solve

\[
P(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0
\]

for coefficients \( a_0, a_1, a_2, a_3 \) by using \( y_i = P(x_i) \), \( i=1,2,3,4 \)

\[
-a_3 + a_2 - a_1 + a_0 = 1
\]

\[
a_0 = 0
\]

\[
a_3 + a_2 + a_1 + a_0 = 0
\]

\[
27a_3 + 9a_2 + 3a_1 + a_0 = 2
\]

Since \( a_3 = -1/24, a_2 = 1/2, a_1 = -11/24, a_0 = 0 \) graph of \( P(x) \) looks like:
In general $P(x)=a_n x^{n-1} + \ldots + a_1 x + a_0$ for $n$ points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ can be obtained by solving the following linear system of $n$ equations in $n$ unknowns $a_{n-1}, \ldots, a_1, a_0$, by using $y_i= P(x_i)$, $i=1,2,3, \ldots, n$

\[
\begin{align*}
    a_{n-1} x_1^{n-1} + \ldots + a_1 x_1 + a_0 &= y_1 \\
    a_{n-1} x_2^{n-1} + \ldots + a_1 x_2 + a_0 &= y_2 \\
    &\vdots \\
    a_{n-1} x_n^{n-1} + \ldots + a_1 x_n + a_0 &= y_n
\end{align*}
\]

The determinant of this system is known as Vandermonde determinant and it is nonzero, and this fact proves that there is a unique solution for this system.
Lagrange Interpolation

The following explicit formula was given by Lagrange for interpolating $P(x)=a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ for $n$ points $\{(x_1,y_1), (x_2,y_2), \ldots, (x_n,y_n)\}$

$$P(x) = \sum_{i \leq i \leq n} \left( \prod_{i \neq j} \frac{(x-x_j)}{(x_i-x_j)} \right) y_i$$

For $n=3$

$$P(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} y_1 + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} y_2 + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} y_3$$

A naive algorithm that simply translates this formula into pseudocode has $\Theta(n^3)$ complexity. Moreover this algorithm can be modified to obtain $\Theta(n^2)$ complexity. HWLA???
Newtonian Interpolation

This method is based on a recursive formula, which expresses the interpolating polynomial \( P_k(x) \) for the \( k \) points \((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\) in terms of the interpolating polynomial \( P_{k-1}(x) \) for the \( k-1 \) points \((x_1, y_1), (x_2, y_2), \ldots, (x_{k-1}, y_{k-1})\). Let \( Q_k(x) = (x-x_1)(x-x_2)\ldots(x-x_k) \).

The interpolating polynomial \( P_k(x) \) satisfies the recurrence relation:

\[
P_k(x) = (y_k - P_{k-1}(x_k)) \frac{Q_{k-1}(x)}{Q_{k-1}(x_k)} + P_{k-1}(x), \quad 2 \leq k \leq n, \quad P_1(x) = y_1
\]
Example for Newtonian Interpolation

Using our previous set of points

\((-1,1),(0,0),(1,0),(3,2)\) = \((x_1,y_1), (x_2,y_2), (x_3,y_3), (x_4,y_4)\),

a sample calculation is as follows;

**Step 1:** \(P_1(x) = y_1 = 1\)

**Step 2:** \(P_2(x) = (y_2 - P_1(x_2))\)

\[\frac{Q_1(x)}{Q_1(x_2)} + P_1(x) = -x\]

**Step 3:** \(Q_2(x) = (x-x_2)Q_1(x) = (x-0)(x-(-1)) = x^2 + x\)

\[P_3(x) = (y_3 - P_2(x_3))\]

\[\frac{Q_3(x)}{Q_3(x_3)} + P_3(x) = (x^2 - x)/2\]

**Step 4:** \(Q_3(x) = (x-x_3)Q_2(x) = (x-1)(x^2 + x) = x^3 - x\)

\[P_4(x) = (y_4 - P_3(x_4))\]

\[\frac{Q_4(x)}{Q_4(x_4)} + P_4(x) = (-1/24)x^3 + (1/2)x^2 + (-11/24)x\]

*And this result agrees with our previous result found by linear algebra.*
Algorithm for Newtonian Interpolation

procedure NewtonInterp(X[1:n],Y[1:n],n,P(x))

Input: X[1:n], Y[1:n] (arrays of real numbers), n (a positive integer)
Output: P(x)=a_{n-1}x^{n-1} + ... + a_1 x + a_0 (interpolating polynomial)

P(x) := Y[1] \{P(x) is initialized to be the constant polynomial y_1\}
Q(x) := x - X[1] \{Q(x) is initialized to be the linear polynomial x - x_1\},

for i := 2 to n do
    PEval := HornerEval(P(x),X[i])
    QEval := HornerEval(Q(x),X[i])
    if i<n then
        Q(x) := PolyMult(Q(x),x-X[i])
    endif
endfor
end NewtonInterp

HornerEval function evaluates a polynomial of the form
\[a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0,\] in following manner

\[((a_4 x + a_3) x + a_2) x + a_1\) x + a_0