

Independent events do not have to be related to independent physical processes.

Example: the probability that the outcome of a die roll is *even* ($= \frac{3}{6}$) is independent of the event "the outcome is ≤ 4 " ($= \frac{4}{6}$).

The probability of "an even outcome ≤ 4 " is

$$\frac{2}{6} = \frac{12}{36} = \frac{3}{6} \cdot \frac{4}{6}$$

Back to Finding the Repeated Element

Repeat the algorithm again, if we don't succeed.

Each time we make **independent** random choices.

If we repeat the algorithm 100 times, the probability of failure is $(4/5)^{100} < 10^{-9}$.

If we repeat the algorithm $c \log n$ times the probability that the algorithm fails is $< (4/5)^{c \log n} = \frac{1}{n^{c \log(5/4)}} = 1/n$ if $c = 1/\log(5/4)$.

Thus the algorithm succeeds **with high probability (whp)**, i.e., with probability at least $1 - 1/n^{\Omega(1)}$.

Thus the algorithm succeeds in $O(\log n)$ time whp.

Random Variable

Let (\mathcal{S}, Pr) be a discrete probability space.

Let V be a set of values.

A random variable X defined on (\mathcal{S}, Pr) is a function

$$X : \mathcal{S} \rightarrow V$$

Let $\mathcal{E}(r) = \{s \in \mathcal{S} \mid X(s) = r\}$

$$Pr(X = r) = Pr(\mathcal{E}(r)) = \sum_{s \in \mathcal{E}(r)} Pr(s).$$

Two random variables X and Y (defined on the same sample space) are called independent if for all x and y

$$Pr\{X = x \text{ and } Y = y\} = Pr\{X = x\} Pr\{Y = y\}$$

Example 1: In rolling a die, the number that comes up is a random variable.

Example 2: Consider a gambling game in which a player flips two coins, if he gets head in both coins he wins \$3, else he losses \$1. The payoff of the game is a random variable.

Expectation

Definition 1. *The expectation of a discrete random variable X is*

$$E[X] = \sum_{i \in \text{range}(X)} i \Pr(X = i).$$

The expectation (or mean or average) is a weighted sum over all possible values of the random variable.

Example: The expected value of one die roll is:

$$E[X] = \sum_{i=1}^6 i \Pr(X = i) = \sum_{i=1}^6 \frac{i}{6} = 3\frac{1}{2}.$$

Consider a game in which a player chooses a number in $[1, \dots, 6]$ and then rolls 3 dice.

The player wins \$1 for each die the matches the number, he losses \$1 if no die matches the number.

What is the expected outcome of that game:

$$-1\left(\frac{5}{6}\right)^3 + 1 \cdot 3\left(\frac{1}{6}\right)\left(\frac{5}{6}\right)^2 + 2 \cdot 3\left(\frac{1}{6}\right)^2\left(\frac{5}{6}\right) + 3\left(\frac{1}{6}\right)^3 = -\frac{17}{216}.$$

Linearity of Expectation

Theorem 1. For any two random variables X and Y

$$E[X + Y] = E[X] + E[Y].$$

Proof.

$$\begin{aligned} E[X + Y] &= \\ \sum_{i \in \text{range}(X)} \sum_{j \in \text{range}(Y)} (i + j) \Pr((X = i) \cap (Y = j)) &= \\ \sum_i \sum_j i \Pr((X = i) \cap (Y = j)) + & \\ \sum_j \sum_i j \Pr((X = i) \cap (Y = j)) &= \\ \sum_i i \Pr(X = i) + \sum_j j \Pr(Y = j). & \end{aligned}$$

□

(Since we sum over all possible choices of i (j).)

Lemma 1. *If E_1, E_2, \dots, E_k are disjoint events such that $\sum_{i=1}^k \Pr(E_i) = 1$ then for any event B ,*

$$\sum_{i=1}^k \Pr(B \cap E_i) = \Pr(B).$$

Examples:

1. The expectation of the sum of two dice is 7, even if they are not independent.

2. Assume that we flip N coins, what is the expected number of heads?

Using linearity of expectation we get $N \cdot \frac{1}{2}$.

By direct summation we get $\sum_{i=0}^N i \binom{N}{i} 2^{-N}$.

Thus we prove

$$\sum_{i=0}^N i \binom{N}{i} 2^{-N} = \frac{N}{2}.$$

3. Assume that N people checked coats in a restaurant. The coats are mixed and each person gets a random coat.

How many people got their own coats?

It's hard to compute $E[X] = \sum_{k=0}^N k Pr(X = k)$. Instead we define N 0-1 random variables X_i , where $X_i = 1$ iff i got his coat.

$$E[X_i] = 1 \cdot Pr(X_i = 1) + 0 \cdot Pr(X_i = 0) =$$

$$Pr(X_i = 1) = \frac{1}{N}.$$

$$E[X] = \sum_{i=1}^N E[X_i] = 1.$$

Back to Randomized Quicksort

Procedure $Q_S(S)$;

Input: A set S .

Output: The set S in sorted order.

1. If $|S| \leq 1$ then return S , else
- 2.(a) Choose a random element y uniformly from S .
(b) Compare all elements of S to y . Let

$$S_1 = \{x \in S - \{y\} \mid x \leq y\}$$

$$S_2 = \{x \in S - \{y\} \mid x > y\}.$$

(Elements in S_1 and S_2 are in the same order as in S .)

- (c) Return the list:

$$Q_S(S_1), y, Q_S(S_2).$$

Analysis

Let T = number of comparisons in a run of QuickSort.

Theorem 2.

$$E[T] = O(n \log n).$$

Proof. Let s_1, \dots, s_n be the elements of S in sorted order.

For $i = 1, \dots, n$, and $j > i$, define 0-1 random variable $X_{i,j}$, s.t.

$X_{i,j} = 1$ iff s_i is compared to s_j in the run of the algorithm.

The number of comparisons in running the algorithm is

$$T = \sum_{i=1}^n \sum_{j>i} X_{i,j}.$$

We are interested in $E[T]$.

What is the probability that $X_{i,j} = 1$?

s_i is compared to s_j iff either s_i or s_j is chosen as a “split item” before any of the $j - i - 1$ elements between s_i and s_j are chosen.

Elements are chosen uniformly at random \rightarrow elements in the set $[s_i, s_{i+1}, \dots, s_j]$ are chosen uniformly at random.

$$Pr(X_{i,j} = 1) = \frac{2}{j - i + 1}.$$

$$E[X_{i,j}] = \frac{2}{j - i + 1}.$$

$$E[T] = E\left[\sum_{i=1}^n \sum_{j>i} X_{i,j}\right] =$$

$$\sum_{i=1}^n \sum_{j>i} E[X_{i,j}] = \sum_{i=1}^n \sum_{j>i} \frac{2}{j - i + 1} \leq$$

$$2n \sum_{k=1}^n \frac{1}{k} \leq 2nH_n = 2n \log n + O(n). \quad \square$$

Probabilistic Analysis of QuickSort

Theorem 3. *The expected run time of (deterministic) Quicksort on a random input, uniformly chosen from all possible permutation of S is $O(n \log n)$.*

Proof.

Set $X_{i,j}$ as before.

If all permutations have equal probability, all permutations of S_i, \dots, S_j have equal probability, thus

$$Pr(X_{i,j}) = \frac{2}{j - i + 1}.$$

$$E\left[\sum_{i=1}^n \sum_{j>i} X_{i,j}\right] = O(n \log n).$$

□

Randomized Algorithms:

- Analysis is true for **any** input.
- The sample space is the space of random choices made by the algorithm.
- Repeated runs are independent.

Probabilistic Analysis;

- The sample space is the space of all possible inputs.
- If the algorithm is **deterministic** repeated runs give the same output.

Randomized Algorithm classification

A **Monte Carlo Algorithm** is a randomized algorithm that may produce an incorrect solution.

For decision problems: A **one-side error** Monte Carlo algorithm errs only on one possible output, otherwise it is a **two-side error** algorithm.

A **Las Vegas** algorithm is a randomized algorithm that **always** produces the correct output.

In both types of algorithms the run-time is a random variable.

Selection

Input: A set S of n distinct elements, and an integer $1 \leq i \leq n$.

Output: The i -th smallest element in S .

Random-Select(S, i) ($1 \leq i \leq |S|$).

1. If $|S| = 1$ then return S .
2. Choose a random element y uniformly from S
3. Compare all elements of S to y . Let

$$S_1 = \{x \in S \mid x < y\}, \quad S_2 = \{x \in S \mid x > y\}.$$

4. If $|S_1| = i - 1$ then return y
 elseif $|S_1| \geq i$ then return **Random-Select**(S_1, i)
 else return **Random-Select**($S_2, i - |S_1| - 1$);

Correctness and Worst-case runtime

Theorem 4. *The algorithm returns a singleton with the correct value.*

Proof.

By induction on the depth of the recursion.

In each call to $\text{Random-Select}(S', i')$, $i' \leq |S'|$ and the i' largest element in S' is the i largest element in S .

When $|S'| = 1$, it includes the i largest element in S . \square

Run-time

Theorem 5. *The worst-case run-time of the algorithm is $O(n^2)$.*

Proof. In the worst case the size of the set that includes the i -th largest element decreases by one in each iteration. \square

Expected run-time

Theorem 6. *The expected run-time of the algorithm is $O(n)$.*

Proof. Without loss of generality we can assume that in each iteration the i -th largest element is in the larger of the two sets S_1 and S_2 .

Let r.v. $T(n)$ = denote the run-time on a set of n elements.

For each $k = 1, 2, \dots, n$, define indicator r.v.s X_k denoting the event that the set S_1 has exactly $k - 1$ elements.

$$E[X_k] = 1/n$$

$$\begin{aligned}
T(n) &\leq \sum_{k=1}^n X_k (T(\text{Max}[k-1, n-k]) + \alpha n) \\
&= \sum_{k=1}^n X_k T(\text{Max}[k-1, n-k]) + \alpha n \\
E[T(n)] &\leq E \left[\sum_{k=1}^n X_k T(\text{Max}[k-1, n-k]) + \alpha n \right] \\
&= \sum_{k=1}^n E[X_k T(\text{Max}[k-1, n-k])] + \alpha n \\
&= \sum_{k=1}^n E[X_k] E[T(\text{Max}[k-1, n-k])] + \alpha n \\
&= \frac{1}{n} \sum_{k=1}^n E[T(\text{Max}[k-1, n-k])] + \alpha n \\
&\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \alpha n
\end{aligned}$$

We show that $T(n) \leq cn$ for some constant $c > 0$.

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \alpha n \\ &= \frac{2c}{n} \left(\sum_k^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + \alpha n \\ &\leq \frac{3}{4}cn + c/2 + \alpha n \\ &\leq cn \end{aligned}$$

for a suitably large c .

Theorem 7. *If X and Y are independent random variable*

$$E[XY] = E[X] \cdot E[Y],$$

Proof.

$$E[XY] = \sum_i \sum_j i \cdot j \Pr((X = i) \cap (Y = j)) =$$

$$\sum_i \sum_j ij \Pr(X = i) \cdot \Pr(Y = j) =$$

$$\left(\sum_i i \Pr(X = i) \right) \left(\sum_j j \Pr(Y = j) \right).$$

□

□

Linear Time Deterministic Selection Algorithm

Theorem 8. *There is a deterministic algorithm that finds the i -th smallest element in an unsorted array of n elements in $O(n)$ time.*

Select (S, i) - Selects the i -th smallest element in the set S .

1. $n = |S|$. If $n = 1$ then return S .
2. Partition S into $\lfloor \frac{n}{5} \rfloor$ groups of 5 elements each, and a leftover group of up to 4 elements.
3. Find the median of each of the groups, let R be the set of these $\lfloor \frac{n}{5} \rfloor$ values.
4. $y = \text{Select}(R, \lceil \frac{|R|}{2} \rceil)$;
5. Compare all elements of S to y . Let
$$S_1 = \{x \in S \mid x < y\}, \quad S_2 = \{x \in S \mid x > y\}.$$
6. If $|S_1| = i - 1$ then return y
 elseif $|S_1| \geq i$ then return $\text{Select}(S_1, i)$
 else return $\text{Select}(S_2, i - |S_1| - 1)$;

Correctness and Runtime

Theorem 9. *The algorithm returns the correct value.*

Proof. By induction on the depth of the recursion. \square

Run-time

Theorem 10. *The run-time of the algorithm is $O(n)$.*

Proof.

How many elements in S are larger than y , the “median of medians” value computed in step 4 of the algorithm?

Excluding the leftover group, and the group that includes y , in at least half of the remaining groups, there are at least three elements that are $> y$. Thus, at least

$$3\left(\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

in S are greater than y .

Similarly, at least $\frac{3n}{10} - 6$ elements in S are $\leq y$.

Thus, select is called in step 6 with at most $\frac{7n}{10} + 6$ elements.

$T(n)$ = run-time on sets of size n .

$$T(n) \leq T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10} + 6) + \alpha n.$$

We show that $T(n) \leq cn$ for some constant $c > 0$.

$$\begin{aligned} T(n) &\leq c(n/5 + 1) + c(7n/10 + 6) + \alpha n \\ &\leq 9cn/10 + 7c + \alpha n \\ &\leq cn \end{aligned}$$

for $n > 70$ and sufficiently large c . \square

Dynamic Programming

A powerful technique for solving **optimization** problems. Useful when the solution to a problem can be viewed as a result of a sequence of decisions.

For example, the solution to the Knapsack problem can be viewed as a sequence of decisions, namely, deciding the values of x_i , $1 \leq i \leq n$. If x_i 's are restricted to be 0/1, then a greedy solution will not work.

One way to solve such problems is to enumerate all possible decision sequences and pick the best. Dynamic programming often drastically reduces the amount of enumeration by appealing to the **Principle of Optimality**.

The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

Ingredients of Dynamic Programming

1. **Optimal substructure:** The optimal solution is computed from optimal solutions to **independent** sub-problems.

Independent means the solution to one subproblem does not affect the solution to another subproblem of the same problem.

2. **Overlapping** subproblems.

Overlapping means the same subproblem occurs in the solution of many different (sub)-problems. We compute the solution of a subproblem *only once* and then storing it in a table and doing a **look up**. This strategy is called **memoization** — a **top-down strategy**.

3. Write a **recursive** formula to evaluate the optimal value.
4. Solve the recursion in a **bottom-up** manner.

Matrix Multiplication

Given a chain of n matrices: $\langle A_1, A_2, \dots, A_n \rangle$
where matrix A_i has dimension $p_{i-1} \times p_i$, for $i = 1, 2, \dots, n$.

Find a way to paranthesize the product $A_1 A_2 \dots A_n$
which **minimizes** the number of multiplications.

Recursive solution

For $1 \leq i \leq j \leq n$, let $m[i, j]$ be the minimum cost of paranthesizing the product $A_i A_{i+1} \dots A_j$.

$$m[i, j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & i < j \end{cases}$$

A Naive Algorithm

cost(p, i, j)

Input: A sequence of matrix dimensions: $p = \langle p_0, p_1, \dots, p_n \rangle$.

Output: the optimal cost to multiply $A_i \dots A_j$.

1 **if** $i = j$ **return** 0

2 $m[i, j] = \infty$

3 **for** $k = i$ **to** $j - 1$

4 $q = \text{cost}(p, i, k) + \text{cost}(p, k + 1, j) + p_{i-1}p_kp_j$

5 **if** $q < m[i, j]$

6 $m[i, j] = q$

7 **return** $m[i, j]$

Bottom-up Algorithm

Matrix-cost(p)

Input: A sequence of matrix dimensions: $p = \langle p_0, p_1, \dots, p_n \rangle$.

Output: the optimal cost to multiply $A_1 \dots A_n$.

```
1 for  $i = 1$  to  $n$ 
     $m[i, i] = 0$ 
2 for  $l = 2$  to  $n$ 
3     for  $i = 1$  to  $n - l + 1$ 
4          $j = i + l - 1$ 
5          $m[i, j] = \infty$ 
6         for  $k = i$  to  $j - 1$ 
7              $q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
8             if  $q < m[i, j]$ 
9                  $m[i, j] = q$ 
10 return  $m[1, n]$ 
```

Memoization: Top-down Algorithm

```
1 for  $i = 1$  to  $n$ 
2   for  $j = i$  to  $n$ 
3      $m[i, j] = \infty$ 
4 return lookup( $p, 1, n$ )
```

lookup(p, i, j)

```
1 if  $m[i, j] < \infty$  return  $m[i, j]$ 
2 if  $i = j$ 
    $m[i, j] = 0$ 
3 else for  $k = i$  to  $j - 1$ 
4    $q = \mathbf{lookup}(p, i, k) + \mathbf{lookup}(p, k + 1, j) + p_{i-1}p_kp_j$ 
5   if  $q < m[i, j]$ 
      $m[i, j] = q$ 
6 return  $m[i, j]$ 
```

A Problem from Computational Biology

Given two (DNA or protein) sequences $s[1 \dots m]$ and $t[1 \dots n]$ we want the "best" **alignment** between the two sequences.

Example:

$$s = GACGGATTAG$$
$$t = GATCGGAATAG$$

An alignment is an insertion of spaces in arbitrary locations along the sequences so that they end up with the same size.

A space in one sequence should not align with a space in the other. But spaces can be inserted at the beginning or at the end.

An alignment can be **scored** by a scoring scheme. We assume a scoring matrix $score$ where the entry

$score[x, y]$ gives the alignment score for characters x and y .

A best alignment is one which receives the maximum score called the **similarity** — $sim(s, t)$.

Recursive formula

$$\text{sim}(s[1 \dots i], t[1 \dots j])$$

$$= \max \begin{cases} \text{sim}(s[1 \dots i - 1], t[1 \dots j - 1]) + \text{score}(s[i], t[j]) \\ \text{sim}(s[1 \dots i], t[1 \dots j - 1]) + \text{score}(-, t[j]) \\ \text{sim}(s[1 \dots i - 1], t[1 \dots j]) + \text{score}(s[i], -) \end{cases}$$

Bottom-up Algorithm

Input: sequences $s[1 \dots m]$ and $t[1 \dots n]$

Output: $sim(s, t)$

for $i = 0$ to m

$a[i, 0] = i \times score(-, t[1])$

for $j = 0$ to n

$a[0, j] = j \times score(s[1], -)$

for $i = 1$ to m

for $j = 1$ to n

$a[i, j] = \max\{a[i - 1, j - 1] + score(s[i], t[j]),$
 $a[i, j - 1] + score(-, t[j]),$
 $a[i - 1, j] + score(s[i], -)\}$

return $a[m, n]$

0/1 Knapsack

Let $KNAP(l, j, y)$ represent the problem:

$$\text{maximize } \sum_{l \leq i \leq j} p_i x_i$$

$$\text{subject to } \sum_{l \leq i \leq j} w_i x_i \leq y$$

$$x_i \in \{0, 1\}, l \leq i \leq j$$

The Knapsack problem is $KNAP(1, n, m)$. For simplicity, we will assume that all the weights and m are integers.

Let $f_j(y)$ be the optimal solution to $KNAP(1, j, y)$. Then,

$$f_n(m) = \max\{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\}$$

because of the principle of optimality.

And for arbitrary $f_i(y)$, $i > 0$,

$$f_i(y) = \max\{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\}$$

Using, $f_0(y) = 0$ for all $y \geq 0$ and

$f_i(y) = -\infty$ if $y < 0$

we compute $f_n(m)$ in a bottom up manner.

The time complexity is $O(nm)$.

Elements of Dynamic Programming

Optimal substructures: A k -stage optimal solution is computed from $k-1$ -stage optimal solutions.

Overlapping substructures: the same $k-1$ -stage substructure is used in the computation of a number of k -stage substructures.