**A.**

*Backtracking Line Search* is an *Inexact Line Search Method*

**Inexact Line Search Method** is as follows:

- Formulate a criterion that assures that steps are neither too long nor too short.
- Pick a good initial step size.
- Construct sequence of updates that satisfy the above criterion after very few steps.

The algorithm of **Backtracking Line Search**:

1) Given $\alpha_{init} > 0$, let $\alpha(0) = \alpha_{init}$ and $t = 0$;

2) Repeat until $f(x^k + \alpha^{(t)} p^k) \leq f(x^k) + c\alpha^{(t)} \cdot [g^k]^T \cdot p^k$,

i) set $\alpha^{(t+1)} = \tau\alpha^{(t)}, where \tau \in (0, 1)$ is fixed (e.g., $\tau = 1/2$),

ii) increment $t$ by 1.

3) Set $\alpha_k = \alpha(t)$

Note: This method prevents the step from getting too small, but it does not prevent steps that are too long relative to the decrease in $f$.

```
ilogit=function(u) return( 1/(1+exp(-u)));

#set.seed(5)
n=100
p=5
## create the data matrix
X=cbind(1, matrix(rnorm(n*(p-1)), nrow=n, ncol=(p-1)))

## create the true beta
beta.star=rnorm(p, mean=0, sd=1/sqrt(p))

## have all experiments have an index of 30
n.list=rep(30, n)
```

```
## create the vector of success probabilities
pi.list=ilogit(as.numeric(X%*%beta.star))

## create the vector of observed sample proportions
## of success, one for each of the n Binomial experiments
y = rbinom(n=n, size=n.list, prob=pi.list)/n.list




stepsize=function(X, y, minusGrad, b, tau, c)
{
t=1
alpha=1
repeating=T
while(repeating)
{
t=t+1
alpha=tau*alpha
pi.t_left=ilogit(as.numeric(X%*%(b+alpha*minusGrad) ) )
pi.t_right=ilogit(as.numeric(X%*%b ) )
left=sum(n.list*y*log(pi.t_left))+sum(n.list*(1-y)*log(1-pi.t_left))
right=sum(n.list*y*log(pi.t_right))+sum(n.list*(1-y)*log(1-pi.t_right)) + c * alpha * t(
if ( ( as.numeric( left ) <=  as.numeric(right) ) | (t>100))
{
repeating=FALSE
}
}
return(list(alpha=alpha,t=t))
```

```
}


gradient.descent=function(X, y, tol, maxit)
{
X.t.y=crossprod(X, y)

beta=matrix(0, nrow=p , ncol=maxit)
beta[,1]=rep(1,p)

loglikelihood=rep(0,maxit)
HessRank=rep(0,maxit)

pi.t0=ilogit(as.numeric(X%*%beta[,1]))

loglikelihood[1]=sum( y * log(pi.t0 + (1-pi.t0)* 1e-5) ) + sum( (1-y) * log(1-pi.t0 + pi

k=1

iterating=T

while( iterating )
{
  k=k+1

  pi.t=ilogit(as.numeric(X%*%beta[,k-1]))
  ## compute the direction
  minusGrad=c(X.t.y-crossprod(X, pi.t))

  stepsize=stepsize(X=X, y=y, minusGrad=minusGrad, b=beta[,k-1], tau=0.5, c=0.25)
```

```
  t=stepsize$t
  step.size=stepsize$alpha
  add=step.size*minusGrad
  beta[,k]=beta[,k-1]+add
  loglikelihood[k]=sum( y * log(pi.t + (1-pi.t)* 1e-5) ) + sum( (1-y) * log(1-pi.t + pi.

  if( (sum(add^2) < tol) | (k >=maxit))
    iterating=FALSE

}
# b=as.numeric(b)
return(list(b=beta[,k], total.iterations=k, beta=beta[, 1:k], loglikelihood=loglikelihoo
}

fit=gradient.descent(X=X, y=y, tol=1e-5 , maxit=100)
```

**B.**

**Quasi - Newton**

The **Secant** equation is $B_{k+1}s_k = y_k$, where $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$.
The secant method is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function $f$.

**Algorithm**

Select starting poing $x_0 \in dom f$, $H_0$

1) Compute quasi-Newton direction $\Delta x = -H_{k-1}\nabla f_k$;
2) Determine step size;
3) Compute $x_k = x_{k-1} + step.size * \Delta x$;
4) Compute $H_k$

**BFGS** formula is one of the most popular formulae for updating the Hessian approximation $B_k$.

$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$.

Some practical implementations of quasi-Newton methods avoid the need to factorize $B_k$ at each iteration by updating the inverse of $B_k$ , instead of $B_k$ itself.

Apply to the inverse approximation $H_k := B_k^{-1}$:

$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k s_k y_k^T) + \rho_k s_k s_k^T$, where $\rho_k = \frac{1}{y_k^T s_k}$.

Calculation of $p_k$ can be performed by $p_k = -H_k \nabla f_k$.

```
quasi.newton=function(X, y, tol, maxit)
{
X.t.y=crossprod(X, y)

beta=matrix(0, nrow=p , ncol=maxit)
beta[,1]=rep(1,p)

loglikelihood=rep(0,maxit)

pi.t0=ilogit(as.numeric(X%*%beta[,1]))

loglikelihood[1]=sum( y * log(pi.t0 + (1-pi.t0)* 1e-5) ) + sum( (1-y) * log(1-pi.t0 + pi

k=1

iterating=T

while( iterating )
{
  k=k+1

  pi.t=ilogit(as.numeric(X%*%beta[,k-1]))
```

```
  ## compute the direction
  minusGrad=c(X.t.y-crossprod(X, pi.t))

  W=diag(pi.t*(1-pi.t))
  ## compute the direction
  Hess=crossprod(X,W%*%X)
  direction= -solve(Hess) * minusGrad

#   step.size=0.001
#   step.size=1/k^1
  stepsize=stepsize(X=X, y=y, minusGrad=minusGrad, b=beta[,k-1], tau=0.5, c=0.25)
  t=stepsize$t
  step.size=stepsize$alpha

#   HessRank[k]=rankMatrix(Hess)

  add=step.size*direction
  beta[,k]=beta[,k-1]+add
  loglikelihood[k]=sum( y * log(pi.t + (1-pi.t)* 1e-5) ) + sum( (1-y) * log(1-pi.t + pi.

  if( (sum(add^2) < tol) | (k >=maxit))
    iterating=FALSE


}
# b=as.numeric(b)
return(list(b=beta[,k], total.iterations=k, beta=beta[, 1:k], loglikelihood=loglikeliho
}
```